Intelligent Platform Management Interface Implementer's Guide

Draft—Version 0.7

9/16/98

REVISION HISTORY

Revision	Date	Author	Description
0.7	9/14/98	Mari S. Schwartz	Initial release

DISCLAIMERS

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. Intel does not warrant or represent that such use will not infringe such rights.

I^2C is a trademark of Philips Semiconductors. All other product names are trademarks, registered trademarks, or servicemarks of their respective owners.

I^2C is a two-wire communications bus/protocol developed by Philips. IPMB is a subset of the I^2C bus/protocol and was developed by Intel. Implementations of the I^2C bus/protocol or the IPMB bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Intel, Hewlett-Packard, NEC, and Dell retain the right to make changes to this document at any time, without notice. Intel, Hewlett-Packard, NEC, and Dell make no warranty for the use of this document and assumes no responsibility for any error which may appear in the document nor does it make a commitment to update the information contained herein.

*Third-party brands and names are the property of their respective owners.

Nothing in this document constitutes a guarantee, warranty, or license, express or implied. Intel disclaims all liability for all such guaranties, warranties, and licenses, including but not limited to: fitness for a particular purpose; merchantability; not infringement of intellectual property or other rights of any third party or of Intel; indemnity; and all others. The reader is advised that third parties may have intellectual property rights that may be relevant to this document and the technologies discussed herein, and is advised to seek the advice of competent legal counsel, without obligation to Intel.

Intel retains the right to make changes to this document at any time, without notice. Intel makes no warranty for the use of this document and assumes no responsibility for any error that may appear in the document nor does it make a commitment to update the information contained herein.

The server manager firmware may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Currently characterized errata are available on request.

CONTENTS

Figures

Tables

1. INTRODUCTION

1.1 Conventions and Terminology

Table 1, Conventions and Terminology

Term	Definition
ACPI	Advanced Configuration and Power Interface. A specification for advanced configuration and power management control. This is an interface to BIOS and OS, and involves the OS, BIOS, hardware, and System Management Software.
API	Application Program Interface.
asserted	Means "logically true." Active-high (high-true) signals are asserted when in the high electrical state. Active-low (low-true) signals are asserted when in the low electrical state.
"behind"	Implementing a device "behind" a management controller (such as a BMC) means that, viewed from the IPMB bus, the device is NOT on the IPMB bus, but is on the other side of the management controller. It can either be directly under the controller (such as the SDR Repository under BMC) or on the controller's private bus (BMC Private Bus). If it is directly under the controller, it is also managed by the controller. This means that the controller implements all the IPMI commands necessary to communicate with what is "behind" it.
BIOS	In this document, BIOS refers only to System BIOS that controls basic system functions using configuration values stored in NVRAM and battery-backed CMOS or Flash ROM.
BMC	Baseboard Management Controller.
Bridge	The circuitry that connects one computer bus to another, allowing an agent on one to access the other.
BT	Block Transfer. One of the IPMI System Interface implementations.
deasserted	Means "logically false." Opposite of "asserted."
DIMM	Dual-inline Memory Module. Name for the plug in modules used to hold the system's DRAM (Dynamic Random Access Memory).
ECC	Error-Correcting Code. Refers to a group of bits on system RAM that are used to provide a check code that is used to verify memory data integrity. This allows certain classes of memory errors to be corrected, and others to be detected.
Emergency management card	An autonomous computer implemented on an add-on card that connects to the baseboard that provides an out-of-band communication path to system management functions when the main system processor or OS is down.
FPC	Front Panel Controller, a satellite management controller supplied by IPMI Implementer.
FRU	Field Replaceable Unit. A component that can be replaced in the field by service personnel or perhaps by a customer.
Hard Reset	A reset event in the system that initializes all components and invalidates caches. Generates physical resets of the hardware.
I^2C	Inter-Integrated Circuit bus. A multi-master, 2-wire, serial bus used as the basis for the Intelligent Platform Management Bus (IPMB).
ICMB	Intelligent Chassis Management Bus. Provides a standardized interface for platform management information and control between chassis. The ICMB is designed so it can be implemented with a device that connects to the IPMB. This allows the ICMB to be implemented as an add-on to systems that have an existing IPMB.
IERR	Internal Error. A signal from the Intel processor indicating a processor internal error condition.
IPM	Intelligent Platform Management

Table 1, Conventions and Terminology (cont.)

IPM device	Intelligent Platform Management device, a.k.a. management controller. A device that supports IPMI protocol and commands, such as a BMC or FPC. Typically implemented with a microcontroller, it performs certain platform management functions independent of the main system processor(s). Management functions include scanning for over-temperature or over-voltage conditions, system security, managing system power control, and other conditions. Compare with "non-intelligent device."
IPMB	Intelligent Platform Management Bus. Name for the architecture, protocol, and implementation of a bus that supports IPMI. The bus is built on I^2C and provides a communications path between management controllers and other IPMI devices.
IPMI	Intelligent Platform Management Interface. Protocol and command sets for IPM devices (management controllers) to communicate with each other.
ISA	Industry Standard Architecture. Name for the basic "PC-AT" functions of an Intel Architecture computer system.
KCS	Keyboard Controller Style (interface). One of the IPMI System Interface implementations
LUN	Logical Unit Number. Identifies the logical unit number within a management controller. Used as part of IPMI Messages.
Management controller	A device that supports IPMI protocol and commands, such as a BMC or a "satellite management controller." Typically implemented with a microcontroller, it performs platform management functions independent of the main system processor(s). " Compare with "non-intelligent device."
NMI	Non-maskable Interrupt. The highest priority interrupt in the system, after SMI. This interrupt has traditionally been used to notify the operating system fatal system hardware error conditions, such as parity errors and unrecoverable bus errors.
Non-intelligent device	A device on the IPMB or Private I^2C bus that do not understand IPMI commands, such as an FRU Inventory serial EEPROM. Compare with "management controller."
Platform	Refers to the hardware portion of a system. Compare with "server."
Platform Management Firmware	Firmware that resides on server management devices on a platform, such as a BMC or the system Interface. Compare with "System Management Software."
POST	Power On Self Test.
Satellite management controller	A management controller located on various boards and modules within the system but away from the baseboard. For example, chassis management controller, front panel management controller. Compare with "BMC."
SEEPROM	Serial Electrically Erasable Programmable Read Only Memory
SEL	System Event Log. Name for a non-volatile storage area that is used to hold system event log information related to platform hardware events.
Server	Refers to an entire system including hardware, software and firmware. Compare with "platform."
SMI	System Management Interrupt. SMI is the highest priority non-maskable interrupt to Intel processors. An SMI causes the processor to enter SMM (System Management Mode). SMI interrupt handler code resides in a memory space that is only accessible in SMM, and runs independently from, and unknown to the OS.
SMIC	Server Management Interface Chip. One of the IPMI System Interface implementations.
SMM	System Management Mode—a special mode of Intel processors entered via an SMI (System Management Interrupt).
SMS	See System Management Software
System Management Software	System software that handles server management. Runs under the OS. Examples: management applications, service providers. Compare with "Platform Management Firmware."

1.2 Purpose of This Manual

As a companion document for the *IPMI Specification*, this manual provides information on how to use the IPMI Specification and implement IPMI as part of your server management system. Also included is a brief introduction to the key elements of IPMI.

1.3 Intended Audience

This manual is written for hardware, firmware and software engineers involved in implementing the IPMI standard in their baseboards and/or modules connected to the baseboard. Familiarity with PCs and Intel Architecture is assumed. Knowledge of the I^2C bus protocol is helpful but not required.

1.4 Organization of This Manual

Sections 2 - 9	Product description. Descriptions of key features of the IPMI standard.
Section 10 - up to Appendix A	"How to" sections. Information on implementing and extending the IPMI standard.
Appendices	Implementation examples and other details

1.5 Reference Documents

Most of the following documents are available on the IPMI web site at http://developer.intel.com/design/servers/ipmi

1. Intelligent Platform Management Interface Specification Version 1.0 (referred to as the IPMI Specification in this document) © 1997, Intel Corporation, Hewlett-Packard Company, NEC Corporation.
2. Intelligent Platform Management Bus Communications Protocol Specification v1.0, ©1997, 1998 Intel Corporation. This document provides the electrical, transport protocol, and specific command specifications for the IPMB.
3. The I^2C Bus and How to Use It, January 1992, Philips Semiconductors. This document provides the timing and electrical specifications for I^2C busses.
4. IPMB Address Allocation, © 1997 Intel Corporation. This document specifies the allocation of I^2C addresses on the IPMB.
5. Platform Management FRU Information Storage Definition v1.0, © 1997, 1998 Intel Corporation. Provides definitions of individual fields within an FRU, and format of Field Replaceable Unit (FRU) information.

2. SERVER MANAGEMENT SUBSYSTEM OVERVIEW

2.1 What is a Server Management Subsystem?

A server management subsystem is composed of hardware, firmware and software embedded in a server system, for the purpose of autonomous monitoring, recovery, and control of the server's health. This is done independently of the main processors, BIOS, and operating system of the server. The ultimate goal of server management is to lower the end user's total cost of server ownership.

2.2 What is IPMI?

IPMI is a specification for hardware and firmware that monitors and controls a server platform independently of the main processors and System Management Software (SMS). IPMI functions can be accessed through SMS. As a hardware-level interface, IPMI sits at the bottom of a typical management software stack, as illustrated in .

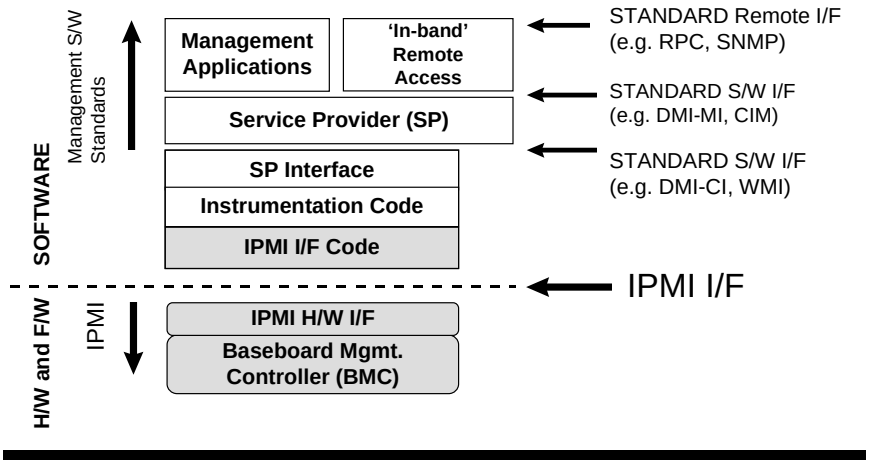


Figure 2-1: IPMI and System Management Software

2.3 Why use IPMI?

- IPMI isolates System Management Software from platform management hardware.
- IPMI is scalable. An IPMI implementation can be cleanly and quickly extended with new functions, such as additional sensors, management controllers, or FRU inventory devices.
- IPMI allows implementers to "plug in" their IPMI implementation to standard System Management Software (SMS) modules.
- IPMI allows the ICMB (Inter Chassis Management Bus—See *Table 1, Conventions and Terminology*) to be implemented as an add-on to systems that have an existing IPMB.
- IPMI supports the WfM (Wired for Management) initiative, which provides a common set of management interfaces to be used by hardware and software vendors. See http://oem.intel.com/ial/wfm/idfwfm.htm for additional information on WfM.

3. IPMI OVERVIEW

This section introduces key elements of IPMI. Please refer to the *IPMI Specification* for detailed information.

The following is an example of IPMI implemented in a typical server.

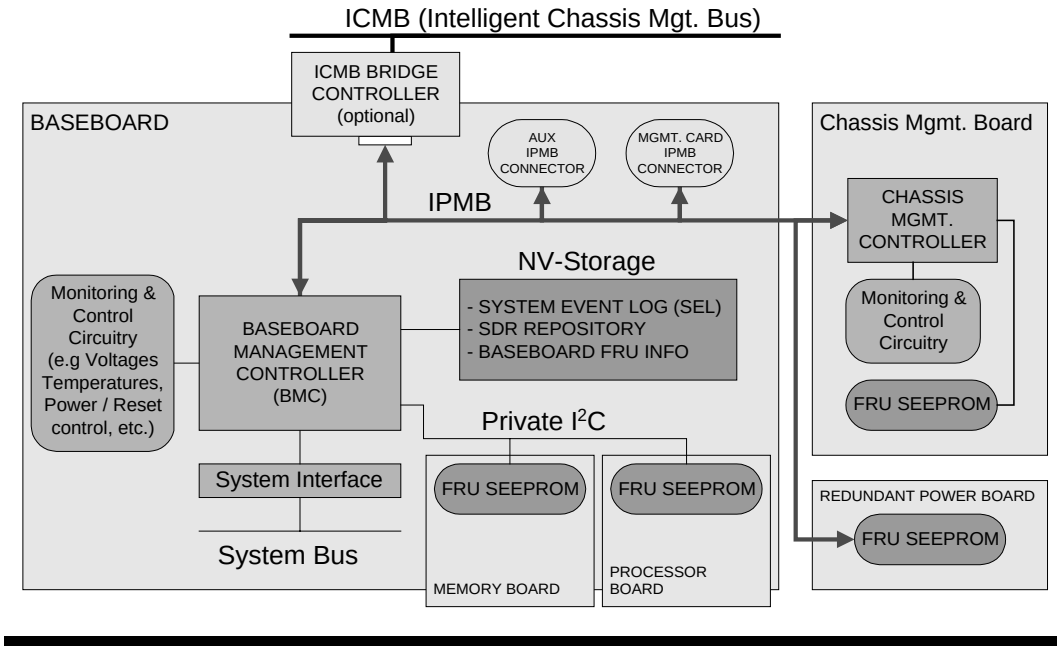


Figure 3-1: IPMI in a Typical Server System

3.1 Management Controllers

Management controllers are microcontrollers on the IPMB that perform platform management functions independent of the main system processor(s). They understand IPMI commands and use them to communicate with each other via the IPMB. There are two types of management controllers: Baseboard Management Controllers (BMC) and satellite management controllers.

3.2 Baseboard Management Controller (BMC)

The BMC is the primary management controller in an IPMI implementation. It typically resides on the baseboard and provides the intelligence behind the IPMI. The BMC manages the interface between System Management Software and platform-management hardware/firmware, provides autonomous monitoring, event logging, and recovery control, and serves as the gateway between System Management Software and the IPMB.

The BMC controls the System Event Log (SEL), Sensor Data Record (SDR) Repository, and BMC FRU (Field Replaceable Unit) and initialization information.

3.3 Satellite Management Controllers

Satellite management controllers are optional and are typically located on various boards and modules within the system but away from the baseboard. For example, the "Chassis Management Controller" in *Figure 3-1: IPMI in a Typical Server System* is a satellite management controller that manages the Chassis Control board.

3.4 Non-Intelligent Devices

Non-intelligent devices, such as FRU SEEPROMs (serial electrically erasable programmable read-only memory) in the above figure, do not understand IPMI commands. They reside directly on the IPMB, or on a private I^2C bus of a management controller.

3.5 Monitoring and Control Circuitry

This circuitry, implemented under the BMC and the Chassis Management Controller in *Figure 3-1: IPMI in a Typical Server System* provides physical connections to sensors that monitor temperatures, voltages, fan status, and other system values. This hardware circuitry is "management software neutral," providing monitoring and control functions that can be exposed through standard management software interfaces such as DMI, WMI, CIM, and SNMP.

3.6 IPMB (Intelligent Platform Management Bus)

The IPMB (Intelligent Platform Management Bus) is an I^2C -based serial bus that is routed between major system modules, including management controllers and non-intelligent devices. The bus integrates satellite management controllers to the baseboard and also connects out-of-band subsystems (such as an emergency management card) to the IPMI. The IPMB is electrically and timing compatible with the I^2C bus specification. Refer to *The I^2C Bus And How To Use It,* under *Section 1.5: Reference Documents.*

3.7 Private Management Busses

A Private Management Bus (also referred to as a Private Bus) is an I^2C bus that is accessed via a management controller by using special IPMI commands for low-level I^2C access. Multiple private busses can be implemented behind a single management controller. IPMI supports private busses as a mechanism for accessing the 24C02-compatible SEEPROMs that hold FRU information. Private busses may also be used to provide low-level access interface for other I^2C devices, though the IPMI specification does not cover the way such devices would be used.

3.8 Communications Between IPMI Devices

Management controllers communicate with each other using IPMI commands over the IPMB. Non-intelligent devices, such as FRU Inventory devices, do not understand IPMI commands, but can be accessed by a management controller using a "low-level" I^2C access command: Master Write-Read I^2C.

3.9 System Interface

This is the interface between the BMC and System Management Software (SMS); it connects the BMC to a system bus that can be driven by the main processor(s). IPMI defines standardized system interfaces that SMS uses for transferring IPMI messages to the BMC. IPMI specifies multiple system interfaces—the SMIC (Server Management Interface Chip), KCS (Keyboard

Controller Style), and BT (Block Transfer) interfaces—allowing a variety of implementations to be used. These system interfaces are similar enough, however, that a single driver can be created to support all IPMI System Interfaces. Refer to the *SMIC Interface, Keyboard Controller Style Interface,* and *Block Transfer Interface* sections of the *IPMI Specification.*

3.10 System Bus

The system interface connects to a system bus that can be driven by the main processor(s). The present IPMI system interfaces are I/O mapped, so any system bus that allows the main processor(s) to access the specified I/O locations and meet the timing specifications can be used. Thus, an IPMI system interface could be hooked to the ISA-bus, X-bus, or a proprietary bus off the baseboard chip set.

3.11 System Event Log (SEL)

The BMC maintains a centralized, non-volatile System Event Log (SEL), which is a repository for system critical events, such as out-of-range temperatures, power supply failures, or uncorrectable ECC errors.

Commands for accessing the SEL can be delivered to the BMC via the IPMB or the System Interface. This makes System Event information accessible from BIOS and System Management Software, as well as out-of-band via a connector on the IPMB. Having the SEL and logging functions managed by the BMC helps ensure that "post-mortem" logging information is available if a failure occurs that disables the system processor(s). Refer to the *Event Messages* section of the *IPMI Specification* for additional information.

3.12 Sensors

Instead of providing direct access to the sensor hardware, IPMI provides access by abstracted sensor commands, such as *Get Sensor Reading* commands, implemented in a management controller. This makes it possible to change sensor and sensor control hardware implementation without affecting software.

Information that is not directly associated with a physical sensor, such as processor status, is also considered a "sensor."

3.13 Sensor Data Records (SDRs) and SDR Repository

The BMC maintains a single, centralized, non-volatile SDR Repository, which holds SDRs (Sensor Data Records) for all sensors in the system. An SDR identifies and completely describes a sensor, including the sensor's default initialization settings that BMC uses at system initialization. An SDR also holds offsets and constants for converting raw sensor readings to standard units (such as milliamps or volts). System Management Software uses the SDR information to get the sensor locations and conversion factors.

The BMC provides an interface to the SDR Repository and allows it to be accessed via the IPMB, the System Interface, "out-of-band" interfaces such as the ICMB, or an add-on emergency-management card. As with most IPMI features, the SDR Repository can be accessed independently of the main processors, the BIOS, and the System Management Software.

The SDR architecture is extensible. It allows the addition of new SDRs to the SDR Repository without in-depth knowledge of the management controller operation. It also allows IPMI implementers to change the sensor's enable/disable states, thresholds, and other parameters.

3.14 FRU Inventory Information

It is recommended that each FRU (field replaceable unit) in the system holds non-volatile FRU Inventory information. FRUs are typically major boards or modules, such as the baseboard, memory boards, processor boards, chassis control boards, power supply, or front panel module.

FRU Inventory information includes serial number, part number, and revision information. It can also have an asset tag, product name, chassis, and other implementation-specific information. Refer to the *Platform Management FRU Information Storage Definition* specification.

4. SYSTEM MANAGEMENT SOFTWARE (SMS) OVERVIEW

This section introduces key elements of the System Management Software, using a DMI-based implementation as an example.

4.1 SMS Functions

The System Management Software runs under the operating system . It handles, interprets, and presents information supplied by the Platform Management Firmware implemented with IPMI.

The System Management Software can provide the following functions in conjunction with Platform Management Firmware.

- Polls for the SEL information, or receives interrupts from firmware indicating a change, and acts on it. System critical events are primarily communicated to SMS using the SEL as a "mailbox" between the event generator and SMS.
- Manages the SEL. Ensures that critical event information is not lost. The SMS is responsible for ensuring that the SEL does not get full.
- Reads and interprets the SDR Repository information. The System Management Software uses this information to determine the sensor population and capabilities.
- Provides access to sensor readings. The System Management Software can use the SDR information to access the sensors, and to monitor and present the system's current health and status.
- Potential Event Message source. The System Management Software can send Event Messages to get events added to the SEL.

4.2 System Management Software Stack

As an example of a typical System Management Software stack, a DMI-based implementation is shown in *Figure 4-1: Example System Management Software/Firmware Stack.* Blocks above the dotted line in the figure are System Management Software components. Blocks under the dotted line are IPMI components previous introduced.

NOTE

For Windows-based management applications, the Management Application and Service Provider blocks in the figure can be replaced with the CIM (Common Information Model) infrastructure. DMI is a temporary solution for Windows-based systems.*

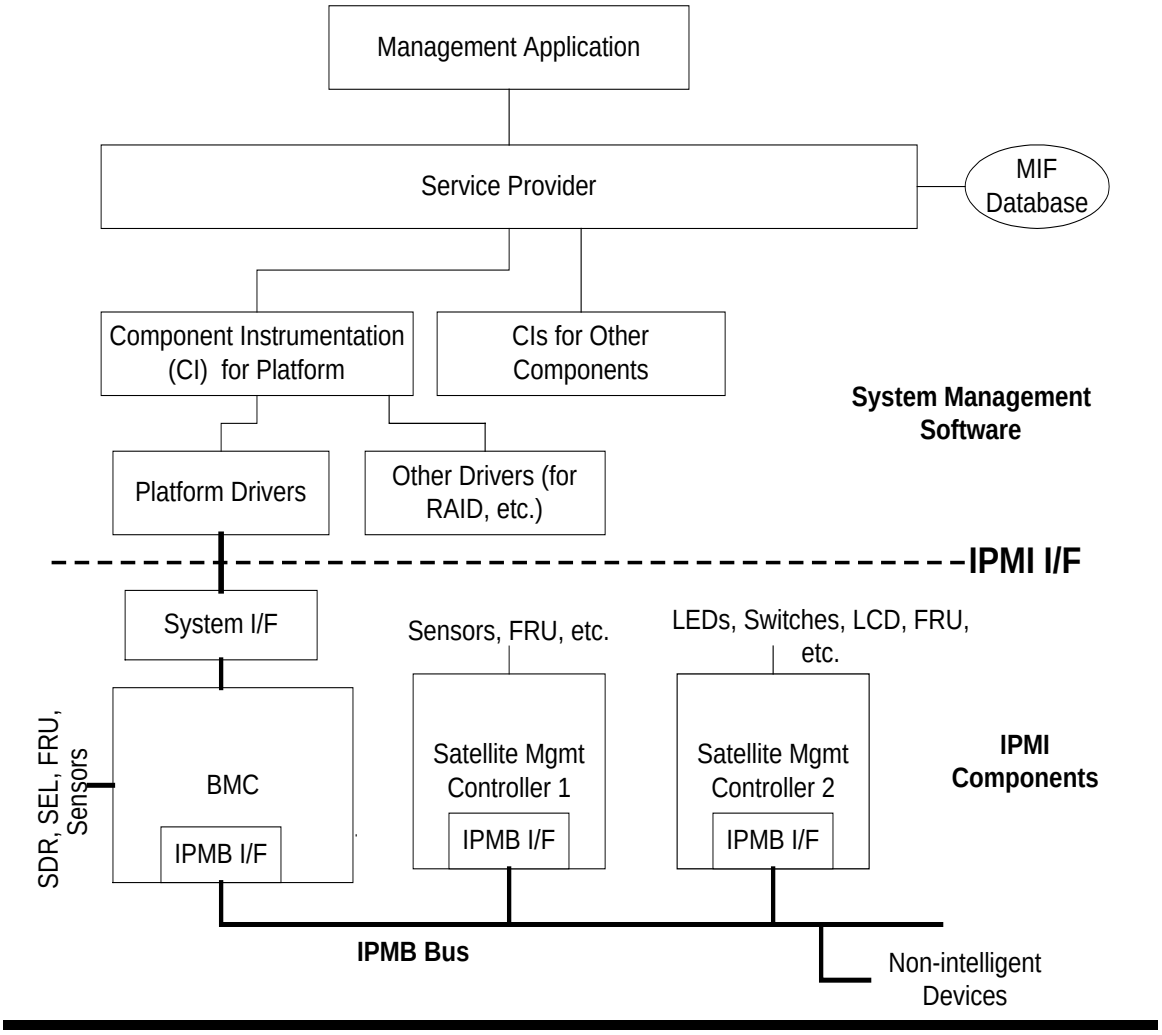


Figure 4-1: Example System Management Software/Firmware Stack

4.2.1 Management Applications

Management Applications run under the operating system and allow remote monitoring and managing of servers and other critical nodes. The primary components of these applications are the managed server (such as, platform) and the monitoring console, (such as, the user interface). Management Applications use the Management Interface to request management activity from components. Examples of Management Applications are HP OpenView Network Node Manager and LANDesk Server Manager.

4.2.2 Service Provider and MIF Database

The Service Provider is a local interface to be used within a single system, which can be a stand-alone desktop, a node on a network, or a networked server. It receives requests from a Management Application and sends them to the appropriate Component Instrumentation (CI). It also collects information from the CI, and passes it to the Management Applications as requested. The Service Provider consists of:

- The Service Provider, a local program that provides information on the type of Component Instrumentation (CI) installed, and what attributes each CI provides. This is done through information in the MIF database (described below). The Service Provider collects information from the components it manages, manages information in the MIF database, and passes the information to Management Applications as requested. The Service Provider controls communication between itself and Management Applications by means of the Management Interface, and between itself and manageable products my means of the Component Interface.
- The MIF database contains information about the products installed on or attached to the system. MIF, the Management Information Format, is the format used by the Service Provider for describing components. The database is the collection of MIF files, which include "templates" for different sensor types.

4.2.3 Component Instrumentation (CI)

CI is the executable code that provides the Service Provider functions for a particular "component" or "platform," such as a processor board set. The CI takes platform information and puts it in a standardized format for the MIF.

4.2.4 Platform Driver

Drivers (OS-level code) provide hardware access to instrumented IPMI devices using one of the System Interfaces defined by IPMI.

4.3 BIOS and IPMI

The level of interaction between BIOS and IPMI is dependent on the implementation. It is possible to have an IPMI implementation that does not require any BIOS support, other than meeting the requirements for the System Interface. Refer to *Section 13: BMC Design Considerations.*

5. EVENT MESSAGES AND ERROR HANDLING

5.1 Introduction

Events and system errors are handled by interactions of the hardware, management controller firmware, BIOS, and System Management Software, as shown in the figure below:

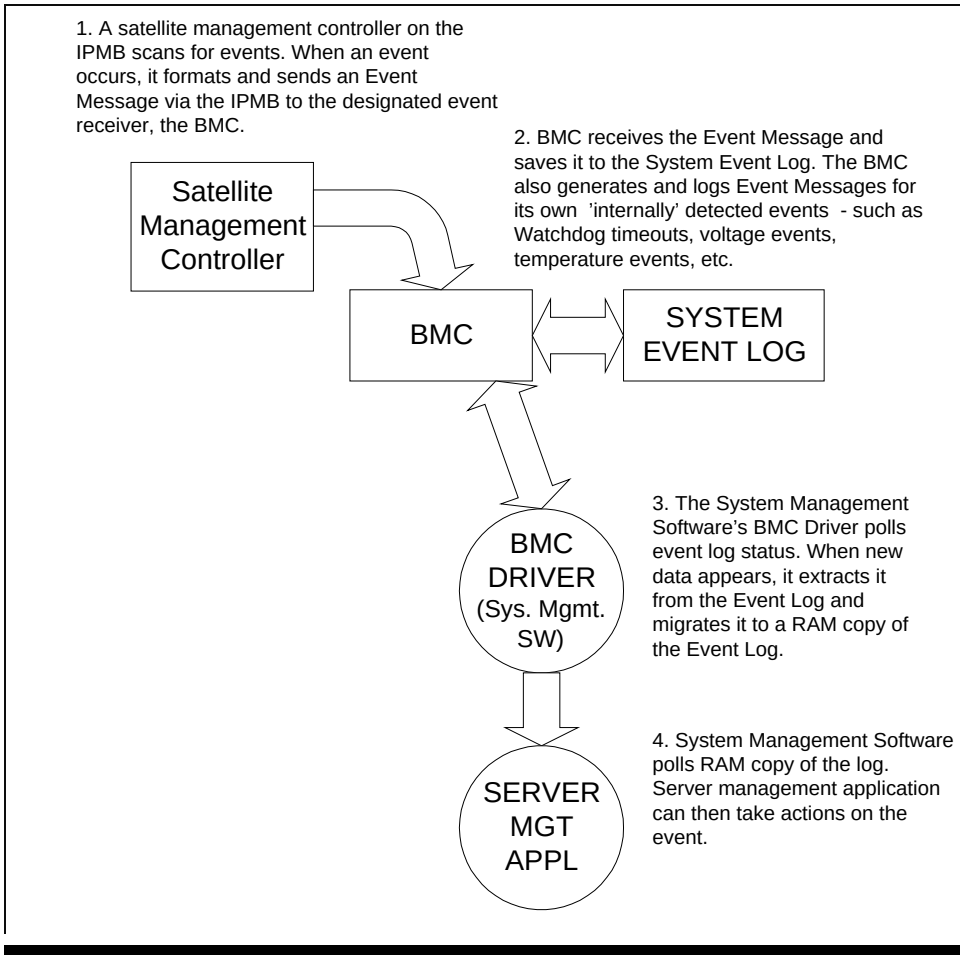


Figure 5-1: Event Message Propagation to System Management Software

5.2 Events and Event Messages

Events are significant or critical system events, such as temperature or voltage thresholds exceeded, power faults, and other failure information that will be used for post-mortem analysis. Also included are impending failures, and events that may require quick system responses, such as system power off or shutdown.

Events are reported as "Event Messages" (a.k.a. "Event Request Messages") by "Event Generators" to the designated "Event Receiver."

NOTE

An "Event Request Message" is a term used by IPMI to distinguish it from an "Event Response Message." In general, the term "Event Message" means an "Event Request Message."

5.3 Event Generators and Receiver

There are multiple "Event Generators," including sensor devices implemented in management controllers, and messaging interfaces of BIOS and the System Management Software. There can be only one "Event Receiver," however, and the power-on default is the BMC. Because it reports events concerning sensors and other functions it directly manages, the BMC can act as both an Event Generator and the Event Receiver.

5.4 Event Message Reception

Upon receipt of a valid Event Message, the Event Receiver (BMC) sends a response message to the Event Generator of the message, then transfers the message to the System Event Log without interpreting the message. Because the BMC does not interpret the Event Messages it receives, new Event Message types can be added into the system without affecting the BMC firmware.

5.5 Event Message Formats

An Event Message, regardless of whether it is received from the IPMB, BIOS, or System Management Software, contains the following fields: Generator ID, Event Message Revision, Sensor Type and Number, and Event Type and Data. An Event Message also includes certain "Codes" in the SDR. For example, when a sensor generates an Event Message, the Event/Reading Type Code and corresponding states (offsets) in the SDR are returned in the Event Message.

There are two slightly different message formats (called the Event Request Message Formats) for messages transmitted on the IPMB bus, and for those generated from the system side by BIOS or System Management Software. Refer to *the IPMI Specification* for detailed information.

5.6 System Event Log (SEL)

Events are logged in the SEL by the BMC as they are received, in the received order, and if requested, delivered up to the System Management Software. Regardless of where the event message comes from, it is logged using the same log format.

Upon receiving an Event Message, the BMC checks it to verify that the message type and checksum are correct. After confirming that it is a valid Event Message, the BMC sends a response message to the generator of the message, timestamps the message, and saves the message in the SEL. The BMC does not interpret the Event Messages it receives. Thus, new Event Message types can be added into the system without impacting the BMC firmware.

The BMC is responsible for filling the SEL, but clearing the SEL once it's full is the responsibility of System Management Software. (The BMC stops logging when the SEL is full.) This guarantees that important event information is not "pushed out" of the log in case a single failure causes a cascade of Event Messages. The philosophy here is that the first "fatal" event is the most important piece of "post mortem" information and that it should not be lost to collect later events.

The format and contents of the SEL, and the BMC commands used to access the log, are specified in the *IPMI Specification.*

6. IPMI HARDWARE COMPONENTS

Table 6-1 describes the components that are required (Req'd) for supporting or enabling IPMI, optional but recommended (Recom), and optional (Opt'l).

Table 5.6-1: IPMI H/W Components—Required, Recommended, and Optional

Item	Req'd	Recom	Opt'l	Hardware Component examples and Note
BMC	x			A microcontroller, such as a Phillips 8xC652 or Hitachi H8-300
IPMB	x	x		A multi-master open-drain serial bus that is electrically and timing compatible with the I^2C bus specification. Refer to the *I^2C Bus and How to Use It.* Both management controllers and "non-intelligent" devices can reside on this bus. IPMB is required if system has multiple management controllers.
Private I^2C Busses			x	These reside "behind" management controllers. Private Busses are single-master I^2C busses under direct control of the BMC or other management controllers. The management controller is always the bus master, and only non-intelligent devices are supported on the bus. Refer to the *I^2C Bus and How to Use It.*
SDR Repository device	x			Any non-volatile ROM. For "Modal" SDR Repository, it can be FLASH, such as the same FLASH that holds BMC code. The Modal SDR Repository can be written only when the BMC is in the repository update mode. Refer to the *IPMI Specification.*
SEL device	x			Any non-volatile storage device. Typically 3 - 8K.

Table 5.6-1: IPMI H/W Components—Required, Recommended, and Optional (cont.)

Satellite Management Controller(s) on IPMB			x	Same as BMC.
FRU Inventory devices		x		May be implemented directly under a management controller, or as a non-intelligent device on the IPMB or a Private I^2C Bus. A 24C02-compatible SEEPROM recommended. If implemented directly under a management controller, any non-volatile storage device may also be used. It is highly recommended that every major module in the system has FRU Inventory information.
System Interface	x			ASIC, standard programmable logic, or keyboard interface built in a microcontroller, depending on the particular interface in your system (SMIC, KCS, or BT). One of these interfaces is required.
IPMB and/or Private I^2C bus remote connection			x	For extending the busses to remote devices and modules. Implement an I^2C signal header, or supply I^2C signals as part of another header.

7. GENERIC MANAGEMENT CONTROLLER FUNCTIONS

This section describes the IPMI functions and commands required for any management controller including the BMC.

7.1 IPMI Functions

The *IPMI Specification* discusses IPMI functions in terms of Logical Devices. Logical Devices are implemented within physical elements, such as management controllers or ASICs. Each Logical Device is implemented with a specific set of IPMI commands.

The Table 7.1-1 summarizes IPMI functions (Device types), associated commands, and whether the functions or commands are Mandatory (M) or Optional (O).

Table 7.1-1: Logical Management Device Types and Applicability

Logical Device Type	**Satellite Management Controller**	**BMC**	**Commands supported by the Device**
IPM Device	M	M	IPM Device "Global" Commands. All management controllers are expected to respond to the mandatory IPM Device commands, such Get Device ID or Get Self Test Results.
Sensor Device	O	O	Sensor Device Commands, such as Get Sensor Reading, Re-arm Sensor Events, or Set Sensor Threshold
SDR Repository Device		M	SDR Repository Commands, such as Get SDR or Add SDR
SEL Device		M	SEL Device Commands, such as Get SEL Entry or Delete SEL Entry
FRU Inventory Device	O	O	FRU Inventory Device Commands, such as Read FRU Inventory Data or Write FRU Inventory Data
Event Receiver Device		M	Event Commands, such as Platform Event (a.k.a. Event Message)
Event Generator Device	O	M	Event Commands, such as Platform Event (a.k.a. Event Message), Set Event Receiver, or Get Event Receiver

Table 7.1-1: Logical Management Device Types and Applicability (cont.)

Application Device	O	O	Commands that are specific to a management controller and are not included in the IPMI specification. They are designed by the implementer, and are considered to be handled by an Application Device.
System Interface		M	This consists of BMC Application Device Commands, called the BMC-System Interface Commands, such as Set/Get BMC Global Enables, Get/Send Message, or Clear/Get Message Flags.

7.2 Optional Functions

If any optional function is implemented, then the mandatory commands for that function are required. For example, if sensors are implemented, then the mandatory Sensor Device commands are required. Refer to *Table 7.4-1: Typical Commands for Generic Management Controllers*.

7.3 Optional but Recommended Functions

For all management controllers that have sensors—the Event Generator Device function is highly recommended. This eliminates the need for System Management Software to poll to detect event conditions.

7.4 1.37. Typical Commands for Generic Management Controllers

The following table summarizes mandatory and recommended commands for all management controllers. The table is provided for quick reference and does not contain all the conditions associated with each command. Refer to the *IPMI Specification* for complete details.

Table 7.4-1: Typical Commands for Generic Management Controllers

Command Set	**Mandatory Commands**	**Recommended (but Optional) Commands**
IPM (Intelligent Platform Management) Commands	Get Device ID Get Self Test Results Broadcast "Get Device ID"	Cold Reset Warm Reset
If controller has sensors "behind" it: Sensor Commands	Get Sensor Reading	Platform Event (a.k.a. Event Message)
If controller generates event messages: Event Commands	Event Message (a.k.a. "Platform Event") Set Event Receiver Get Event Receiver	None
If controller has FRU Inventory "behind" it: FRU Inventory Device Commands	Get FRU Inventory Area Info Read FRU Inventory Data Write FRU Inventory Data	None
If controller has Private Bus "behind" it: BMC-System Interface Commands	Master Write-Read I^2C. Private I^2C Bus is mainly for SEEPROM access, or for other non-intelligent devices you want SMS to access.	None

8. BMC FUNCTIONS

This section summarizes the IPMI functions and commands unique to the BMC. Refer to the *Required BMC Functions* section of the *IPMI Specification* for detailed descriptions.

8.1 Required BMC Functions

The following is a summary of required functions for the BMC.

- All the required functions for generic management controllers, described in *Section 7.2*.
- System Interface
- SDR Repository
- SEL Interface
- Event Receiver
- Internal Event Generation (for generating event messages for the Watchdog Timer that is internal to the BMC)
- Watchdog Timer
- BMC Initialization Agent

8.2 Optional but Recommended BMC Functions

- IPMB Interface (to support a IPMB bus)

8.3 Typical Commands for the BMC

The following table lists mandatory and recommended commands for the BMC. The table is provided for quick reference and does not contain all the conditions associated with each command. Refer to the *IPMI Specification* for complete details.

Table 8.3-1: Typical Commands for BMC

Command Set	**Mandatory Commands**	**Recommended (but Optional) Commands**
IPM (Intelligent Platform Management) Commands	Get Device ID Get Self Test Results Broadcast "Get Device ID"	Cold Reset Warm Reset
BMC-System Interface Commands	Set BMC Global Enables Get BMC Global Enables Clear Message Flags Other commands may be required depending on the IPMI architecture and the particular System Interface used: the SMIC, KCS or BT Interface.	Master Write-Read I^2C Get Message Send Message NOTE: All of the above commands are required if the BMC supports the IPMB.
Event Commands	Ability to receive and acknowledge Event Message Ability to generate Event Message to support the Watchdog Timer. *NOTE: BMC does not support Set/Get Event Receiver, because it is the event receiver*.	None
SDR Repository Device Commands	Get SDR Repository Info Reserve SDR Repository Get SDR Add SDR or Partial Add SDR Clear SDR Repository Exit SDR Repository Update Mode (for systems with "Modal" SDR Repository only)	Run Initialization Agent
SEL Device Commands	Get SEL Info Get SEL Entry Add SEL Entry or Partial Add SEL Entry Clear SEL Get SEL Time Set SEL Time	Delete SEL Entry Reserve SEL
Watchdog Timer Commands	Reset Watchdog Timer Set Watchdog Timer Get Watchdog Timer	None
Init Agent	Get Device ID Set Event Receiver Set Sensor Threshold Set Sensor Hysterisis Set Sensor Event Message Enable	None

Table 8.3-1: Typical Commands for BMC (cont.)

If BMC has FRU Inventory "behind" it: FRU Inventory Device Commands	Get FRU Inventory Area Info Read FRU Inventory Data Write FRU Inventory Data	None
If BMC has sensors "behind" it": Sensor Device Commands	Get Sensor Reading	Platform Event *(This command is required for BMC under Event Commands.)*

8.4 Watchdog Timer

IPMI provides a standardized interface for a system Watchdog Timer that is implemented in the BMC. This timer can be used for BIOS, OS, and IPMI applications. The timer can be configured to automatically generate selected actions when it expires, including power off, power cycle, reset, and interrupt. The timer function can also automatically log the expiration event.

8.5 BMC Initialization Agent

After power-on and system resets, the BMC executes an initialization function called the BMC Init Agent. The BMC Init Agent reads SDR records from the SDR Repository, and writes these data to sensors that have "initialization required" bits set in their SDR. The values that can be set include event generation enables, threshold and hysterisis values, and sensor and event-type information. In other words, the Init Agent uses values in the SDRs to enable sensor events and initialize thresholds. As long as their corresponding SDRs exist, the Init Agent handles all sensors in the entire server management subsystem, including those managed by satellite management controllers.

9. HARDWARE AND BIOS SUPPORT FOR THE BMC

This section describes the functions required of the hardware and BIOS for supporting the BMC.

9.1 Required Features

- The BMC needs to know when the system has powered up or down, or has been given hard reset or a warm "Ctrl-Alt-Del" reset. This information is needed to trigger the BMC Init Agent function.
- The BMC needs to know the system timestamp.

9.2 Recommended Features

- Capability for the BIOS and SMS to send IPMI commands to the BMC. For example, send commands requesting the BMC to log system events in the SEL.

9.3 Optional Add-on Value-Add Features

Additional hardware, SMS and BIOS changes that can be provided for add-on features such as those listed below. These features are not part of IPMI but can take advantage of management controllers. Refer to *H: Design Considerations for Add-on Optional Features*.

NOTE: These features require corresponding functions in the BMC.

- ACPI support.
- Standby (always "on") +5V. Required if you implement an out-of-band emergency management card.
- SMI (System Management Interrupt) Handler in the BIOS. Ability in the handler to send commands to the BMC to log SMI events in the SEL. SMIs may be used for handling additional platform errors or events.
- Security features, such as chassis intrusion or keyboard lockout.

10. IPMI System Design Considerations

This section provides information that will be helpful when designing an IPMI-based system. Additional information specific to satellite management controllers and the BMC are provided in *Section 11*, and *Section 12*.

10.1 Storage Requirements

10.1.1 SDR Repository

- Determine the memory size by the number of sensors in your system plus 20% for expansion (recommended).
- An SDR Type 02h (Compact Sensor Record) saves space, requiring 48 bytes per SDR. An SDR Type 01h (Full Sensor Record) requires 64 bytes.
- Example: A mid-range processor server with 2 processors:

SDR Type	Description	Total count
01h	Full Sensor Record	24
02h	Compact Sensor Record	8
10h	Generic Device Locator Record	2
C0h	OEM SDR	1

10.1.2 SEL (Sensor Event Log)

- The size of the SEL is a function of how many events you expect to receive before the System Management Software can process the log.
- The minimum requirement is 16 entries; each entry has 16 bytes.
- Typical SEL size is 3 - 8K.

10.1.3 FRU Inventory

- The FRU Inventory size varies by implementation. It's affected by the size of the Internal Use Area, which stores non-volatile parameters private to management controllers, such as pointers to SDR and SEL and other parameters. The size of this area varies by what is required by a management controller for IPMI and OEM-specific functions.
- A typical BMC may require over 256KB. A satellite management controller may require less.

10.2 The Number of Devices IPMB Can Support

Depending on the type of device, I^2C addressing places a limit on the number of devices that can be placed directly on the IPMB. Refer to the *IPMB I^2C Address Allocation* specification for more information.

10.3 Check List for IPMI System Design

The following are typical questions and some answers for your consideration during design:

10.3.1 For Baseboard Designers

- ❑ You need to build the BMC.
- ❑ Select a microcontroller for the BMC.
- ❑ Is a good emulator available for the microcontroller?
- ❑ Do you need extensibility of your system? If YES, implement the IPMB.
- ❑ Do you have many chassis to monitor? If YES, we recommend you implement a satellite management controller (SMC) for chassis management.
- ❑ Select a microcontroller for the SMC.
- ❑ Is a good emulator available for the microcontroller?
- ❑ Is there pre-existing driver software your system must be compatible with?
- ❑ Is compatibility with cross-platform System Management Software important or required?
- ❑ Do you need satellite management controllers? How many?
- ❑ Will your system have sensors? How many? What type? What location?
- ❑ What type of sensors are supported by the target System Management Software? Do you plan to write your drivers and other software?
- ❑ How many FRU Inventory devices do you need in the system? What locations? Whether or not they are "behind" a management controller or implemented as non-intelligent devices affects the hardware components you can use.
- ❑ Does your system require any of the optional IPMI commands? Which ones?
- ❑ Does your system support any of the optional add-on features? This will probably require additional IPMI commands.
- ❑ Do you need provisions for testability?
- ❑ Identify tools and utilities you need for development and maintenance. Refer to *Tools and Utilities*.

10.3.2 For Designers of Modules that Connect to the Baseboard

- ❑ How much management functions does your module require? If it requires more than supporting an FRU Inventory, you need a satellite management controller.
- ❑ Is the module's management intimately tied to the baseboard? If YES, add a satellite management controller (SMC).
- ❑ Select a microcontroller for the SMC.
- ❑ Is a good emulator available for the microcontroller?

- ❑ Does the module have sensors? How many? What type? What location?
- ❑ Do you need provisions for testability?
- ❑ Identify tools and utilities you need for development and maintenance. Refer to *Tools and Utilities*.

10.4 Relative priorities of IPMB devices

We recommend the following:

- The IPMB Interface has a higher priority than the System Interface.
- The SMI Handler, if one exists, has a higher priority than the IPMB Interface and System Interface.20.

11. How to Implement a Satellite Management Controller

This section contains specific design information for satellite management controllers. Refer also to *Figure 4-1: Example System Management Software/Firmware Stack.*

11.1 Check List for Designing a Satellite Management Controller (SMC)

- ❑ Select a microcontroller for the SMC.
- ❑ Is a good emulator available for the microcontroller?
- ❑ Assign a device address to the SMC, according to the *IPMB Address Allocation* document. You can generally use the "c" (for chassis vendor use), "b" (for board set vendor use) or "a" (for third-party add-ins) addresses. For example, if the controller manages RAID, backplane, or other chassis-related functions, use a "c" address from the document.
- ❑ Implement the mandatory commands of the IPM Device "Global" Commands set, such as Get Device ID, Get Self Test Result, or Broadcast Get Device ID.
- ❑ Implement optional commands from the "Global" Command set as desired. Refer to the *IPM Device "Global" Commands* section of the *IPMI Specification*.
- ❑ Implement commands from other command sets, depending on the function of the management controller. For example:

If the management controller...	Implement these commands
... is an Event Generator	mandatory Event commands
... has sensors under it	mandatory Sensor Device commands. Refer to the *How to Implement Sensors and SDRs* section for additional information.
... has an FRU Inventory device accessed via FRU Inventory Device commands	mandatory FRU Inventory Device commands
... has an FRU Inventory device on the controller's private bus	Master Write-Read I^2C command

- ❑ Implement an SDR Type 12h record (Management Controller Device Locator) for the management controller.
- ❑ Depending on your design requirements, also implement an SDR Type 13h (Management Controller Confirmation) record. This can be used to record that a specific management controller has been discovered in the system. Later, the record information can be used to confirm that the same controller is still present.
- ❑ Select the location of FRU Inventory. Do you need to access them using the FRU Inventory Device commands? If so, implement them directly "behind" the management controller. If you need to implement the FRU using SEEPROM, implement a private I^2C bus behind the management controller.

- ❑ If you have sensors, select sensor hardware components. Sensor hardware devices are implementer's choice.
- ❑ If you have sensors, select their locations.
- ❑ If you have sensors, decide how the sensor hardware devices are connected to the system.
- ❑ Implement additional application-specific commands as desired.

12. How to Implement the BMC

This section contains design information specific to the BMC. This section covers the BMC, BMC Init Agent, SDR Repository, and SEL. Refer to *Figure 4-1: Example System Management Software/Firmware Stack.*

12.1 BMC Design Considerations

BMC design requires decisions concerning the BMC itself, non-volatile storage (SEL, SDR Repository, FRU Inventory, BMC code), System Interface, IPMB support, and support of any add-in value-add features. The following is a list of typical questions you may want to consider:

- Select a microcontroller for the BMC.
- Is a good emulator available for the microcontroller?
- Select the System Interface, considering tradeoffs. Your decision may affect the hardware component you use for the BMC. Refer to *Table 12.1-1: System Interface Comparison* as well as the *IPMI Specification.*
- Will the BMC have sensors "behind" itself? How many? What type? This will directly affect your command requirements, the size of the SDR Repository and BMC code size.
- Are the types of sensors you need supported by target cross-platform System Management Software, or do you plan to write your drivers and other software?
- Bus architecture: If your system has at least one satellite management controller, or it requires extensibility, then implement the IPMB.
- Do you need a Private I^2C bus for the BMC to support a non-intelligent FRU Inventory on another module, such as a memory module?
- Does your system have an FRU Inventory for the baseboard? If YES, is the FRU Inventory device "behind" the BMC? If so, the BMC needs to support the FRU Inventory Device Commands. NOTE: The "primary" FRU for the baseboard is always accessed as Device 0 at LUN00 of the BMC.
- Storage: A SDR Repository and SEL are mandatory.
- SEL implementation: Refer to *Section 10.1: Storage Requirements*.
- SDR implementation: Refer to *Section 10.1: Storage Requirements*.
- Non-volatile Storage for the BMC: How many storage devices do you need to hold the SDR Repository, SEL, FRU Inventory, and BMC code? What type of devices?—FLASH, SEEPROM, battery-backed storage, or other?
- What system functions does the BMC require in the BIOS and/or System Management Software?
- Do you have functions not covered by mandatory commands? If so, check to see if an IPMI optional commands cover this. Otherwise, consider defining a BMC-specific application commands.

- ❑ Does the BMC support any of the optional add-on features? This may require additional commands.
- ❑ What tools and utilities do you need for development and maintenance? Refer *Tools and Utilities*.

Table 12.1-1: System Interface Comparison

System Interface	Transfer type	Advantages	Disadvantages
KCS (Keyboard Controller Style)	byte transfer	- KCS is available as a built-in interface some microcontrollers, eliminating the need for a separate device. - Supports polled operation - Supports interrupts as an option	
SMIC (Server Management Interface Chip)	byte transfer, I/O mapped	- Low cost option for using microcontroller without built-in KCS interface - Supports polled operation - Supports interrupts as option - Can be shared by system software running under the OS and running within an SMI Handler	Typically requires additional device.
BT (Block Transfer)	block transfer	- Higher performance - Supports polled operation - Supports interrupts as option - Supports submission and asynchronous completion of commands	- Requires additional device - Requires more logic than SMIC interface

12.2 Check List for BMC Design

- ❑ Assign device address 20h to the BMC. Refer to the *IPMB Address Allocation* document.
- ❑ Implement the LUN 00b in the BMC. LUN 00b is required for responding to a Get Device ID command, which is one of the mandatory commands for management controllers.
- ❑ Implement the mandatory commands of the IPM Device "Global" Commands set, such as Get Device ID, Get Self Test Result and Broadcast "Get Device ID."
- ❑ Do you have functions not covered by mandatory commands? If so, check to see if an IPMI optional commands cover this. Otherwise, consider defining a BMC-specific application commands.
- ❑ Does the BMC support any of the optional add-on features? This may require additional commands.

- ❑ Implement an SDR Type 12h record (Management Controller Device Locator).
- ❑ Implement an SDR Type 14h (BMC Message Channel Info). This record describes the allocation and type for the BMC message channels. Refer to the *SDR Type 14h* section of the *IPMI Specification.*
- ❑ Implement the BMC Init Agent. Refer to *Section 12.3: How to Implement the BMC Init Agent.*
- ❑ Implement the FRU Inventory. The primary FRU for the baseboard must be implemented as Device 0 at LUN0 of the BMC, and must be accessible using Read/Write FRU commands.
- ❑ In the management controller's SDR (Type 12h), make sure the Device Support bit for FRU is set in the Device Capabilities field.
- ❑ Implement sensors. Refer to later sections for additional information.
- ❑ Implement SDR Repository. Refer to later sections for additional information.
- ❑ Implement SEL. Refer to later sections for additional information.

12.3 How to Implement the BMC Init Agent

12.3.1 Overview

The BMC Init Agent is a required feature of IPMI. It runs as part of power-on and reset initialization, and restores "power-on default" threshold values and sensor event enable settings.

The Init Agent reads SDR records from the SDR Repository and, using that data, initializes and enables corresponding sensors in the entire system, including those managed by the BMC and satellite management controllers.

The Init Agent reads SDRs in two passes. The first pass is for making a list of management controllers on the IPMB. The second pass is for sensor initialization.

Please refer to the *Sensor Initialization Agent* section of the *IPMI Specification* for complete information on this topic.

12.3.2 System Software Support Required for the BMC Init Agent

The BMC needs to know when the system has been powered up, hard reset, or warm "Ctrl-Alt-Del" reset. This information is needed to trigger the Init Agent. The mechanism for accomplishing this is implementation-dependent. Two common ways to provide this information are via hardware signals to the BMC, or via a BMC-specific application command from BIOS. A combination of the two can also be used. For example, hardware signals could be used to indicate when the system is hard-reset, while a command from BIOS could indicate warm "Ctrl-Alt-Del" resets.

12.4 How to Implement the SDR Repository

- ❑ Decide if you want a "Modal" or non-Modal SDR Repository. Refer to the *IPMI Specification.*
- ❑ Implement the Repository "behind" the BMC with non-volatile storage device of your choice. For "Modal" SDR Repository, it can be FLASH. It could be the same FLASH that holds BMC code.

- ❑ Decide if you want the SDR Repository, SEL, and FRU Inventory in the same storage device or different devices.
- ❑ Implement the mandatory SDR Repository Commands in the management controller (BMC). Optional commands may be used to optimize utility operation, but we recommend that utilities work with just mandatory commands. Refer to *Section 18* for additional information and utility requirements.
- ❑ In the management controller's SDR (Type 12h), make sure the Device Support bit for SDR is set in the Device Capabilities field.

12.5 How to Implement the System Event Log (SEL)

12.5.1 Design Considerations

- Determine the storage size. It is a function of the number of events the SEL is expected to receive before System Management Software can process the event log.
- The minimum requirement is 16 entries, each entry being 16 bytes.
- Any non-volatile storage device may be used. The selection is implementation dependent. A typical size is 3 - 8K.

Refer to the *Event Messages* section of the *IPMI Specification* for additional information, such as:

- The types of events that should be saved in the SEL
- What are critical events
- How the BMC handles events it receives and how it saves them in the SEL
- Advice for implementing the BMC's Event Receiver device

12.5.2 How to Implement the SEL

- Implement the SEL with a non-volatile storage device of your choice.
- Implement the mandatory SEL Commands in the management controller (BMC).
- In the management controller's SDR (Type 12h), make sure the Device Support bit for SEL is set in the Device Capabilities field.

12.5.3 How to Enable/Disable Event Logging

To enable or disable logging of Event Messages into the SEL, you can use the BIOS System Setup Utility. The normal default is that event logging is enabled.

13. HOW TO IMPLEMENT AN SDR

An SDR must be defined for sensors, management controllers, non-intelligent devices on the IPMB, and FRU devices except for FRU devices accessed using Read/Write FRU commands at Device 00, LUN00. These are covered by the Management Controller Device Locator Record. Different SDR Types are required depending on the particular device. For example, a sensor requires an SDR of Type 01h or 02h (Sensor Data Records).

- ❑ Design individual SDRs for the entire system.

1) Determine the SDR Type Code for your device. Refer to *Table 12.5-1: Sensor Data Record (SDR) Types*.

2) Go to the appropriate SDR Type section of the *IPMI Specification,* such as *SDR Type 02h, Compact Sensor Record.*

3) Follow descriptions in the SDR format table and select values for the individual parameters of the SDR.*:*

- ❑ Load all the SDRs to the SDR Repository. Refer to *Section 18*.

Table 12.5-1: Sensor Data Record (SDR) Types

SDR Type Code	Name	Required for:	Description
01h	Sensor (Full Sensor Record)	Sensor (use 01h or 02h)	Applies if sensor returns analog readings, or generates threshold-based events or status, or requires settable hysterisis.
02h	Sensor (Compact Sensor Record)	Sensor (use 01h or 02h)	Compact Sensor Record. Sensors that return "digital" or discrete readings, and do not have threshold-triggered events.
08h	Entity Association	Used to indicate the relationship where a physical or logical entity is comprised of multiple physical or logical entities.	Used for situations like when a Power Unit consists of multiple Power Supply entities. See the *SDR Type 08h* section of the *IPMI Specification.*
10h	Generic Device Locator	Devices on IPMB or private busses that are neither IPMI FRUs nor management controllers.	Used for non-intelligent I^2C devices other than FRU SEEPROMs, special management ASICs or proprietary controllers, and accessible via Master Write-Read I^2C command.
11h	FRU Device Locator	FRU Info directly on IPMB or private bus.	Not applicable to FRU information accessed via FRU commands at FRU Device 00 at LUN 00b of a management controller.
12h	Management Controller Device Locator	Management controllers, including BMC	Used to identify management controllers.

Table 12.5-1: Sensor Data Record (SDR) Types

13h	Management Controller Confirmation (OPTIONAL)	Same as above	Can be used to record discovery of a particular management controller by System Management Software. Can also be used by SMS to confirm that the same controller is still present.
14h	BMC Message Channel Info (OPTIONAL)	BMC	Describes the allocation and type for BMC message channels.
C0h	OEM SDR	OEM defined	OEM defined SDR.

13.1.1 How to Modify Existing SDRs in SDR Repository

The universal method is to have utilities read the contents of SDR Repository, update them, clear the Repository, then write the updated contents back in. In addition, you may implement a utility that uses optional commands, but it is recommended that utility function with just mandatory commands also. Refer to *Section 18.*

14. How to Implement an FRU Inventory Device

14.1 Implementation Options

IPMI FRU Inventory devices may be implemented in different ways at different locations as in the following figure:

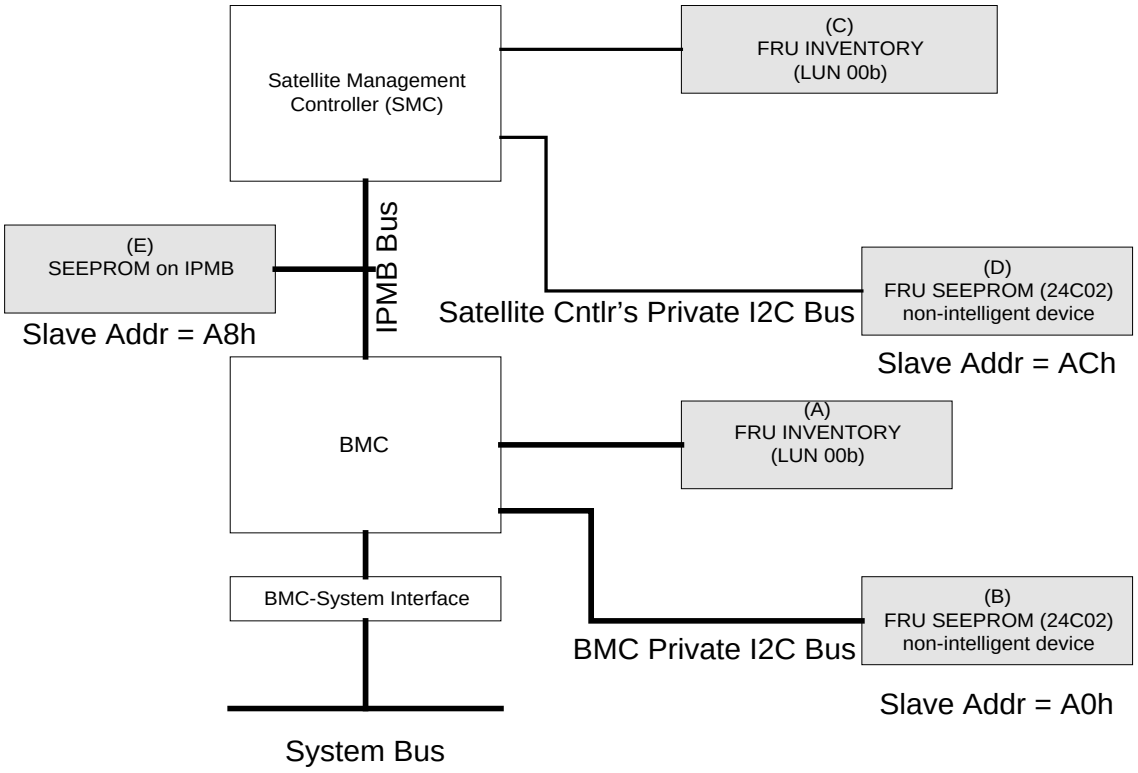


Figure 14-1: Different Implementations of FRU Inventory

Table 14.1-1: FRU Implementation Options

FRU Inventory Location	**Typical Examples**	**Commands Required for Access**	**Advantages (+) and Disadvantages (-)**	**Recommended Use**
"Behind" management controller and accessed by Read/Write FRU commands (A) (C)	(A) — FRU for the baseboard (C) – FRU for chassis modules, such as a RAID, backplane	FRU Inventory Device Commands	+ Any hardware can be used for FRU Inventory device - Needs management controller for access	When mgmt. Controller is on same FRU, or mgmt. Controller has a priori knowledge of (or way of discovering) FRUs for other modules.
On a Private I^2C Bus (B) (D)	(B) —For FRUs on modules attached to the baseboard, such as memory or CPU cards. (D) – For modules that plug into chassis, such as a power distribution board	Master Write-Read I^2C command in management controller that owns Private I^2C Bus.	+ Commands are encapsulated in IPMI messages, so data integrity check is performed during transmission. + Requires no f/w change in controller when the number of FRUs on a Private Bus changes. - Hardware limited to 24C02 family SEEPROM - Need a management controller for access - Master Write-Read I^2C command required in the management controller	Use this when you have FRU devices that may or may not be present, and you do not want to implement firmware in your controller for discovering these devices. (This method allows you to add or remove FRU devices from Private Bus without involving firmware.)

Table 14.1-1: FRU Implementation Options

Directly on the IPMB (E)	This type is not recommended. It may be used for a front panel card or other major modules in the chassis.	"Raw" Master Write-Read I^2C command from BMC. These are raw I^2C operations on the IPMB, and commands are not encapsulated in IPMI messages.	+ Allows adding FRUs without adding a satellite management controller. (Good for chassis FRU, for example.) -Hardware is limited to the 24C02 family SEEPROM - Limited addressing support. - Only four standard addresses for the baseboard and four for the chassis. - Raw I^2C operations have some potential degrading data integrity during transmission. - Need to rely on checksum within FRU information.	Use this when you have a FRU device, but you don't want to use a management controller, or don't have a private bus available.

14.1.1 FRU Inventory "Behind" a Management Controller

The FRU Inventory is accessed through a management controller using FRU Inventory Device commands. The management controller may be the BMC or a satellite management controller.

This is the preferred method, because no custom firmware or software is required. Because this implementation isolates software from direct access to the FRU Inventory storage, you can use any non-volatile storage you want, such as EEPROM or flash ROM. Implementation requires the following:

- ❑ Implement the FRU Inventory in a non-storage device of your choice.
- ❑ Implement the FRU Inventory Device commands in the appropriate management controller. The commands assume that the FRU is in LUN00. (No SDR is required for FRUs implemented this way.) NOTE: The "primary" FRU for the baseboard is always accessed as Device 0 at LUN00 of the BMC.
- ❑ In the management controller's SDR (Type 12h), set the Device Support bit for the FRU in the Device Capabilities field.
- ❑ Design the content of the FRU Inventory. Refer to *Section14.2: FRU Inventory Area Format.*
- ❑ Create a load file for the FRU Inventory and load it to the device, using particular utility.

14.1.2 IPMI FRU on a Private I²C Bus

In this implementation, the FRU Inventory device must be implemented with 24C02-compatible SEEPROMs (non-volatile storage devices with built-in I²C -compatible interface and provisions for checksums of data).These devices are referred to as the "IPMI FRU devices" in the *IPMI Specification.* They are accessed using the low-level I²C access command Master Write-Read I²C encapsulated in an IPMI message.

Implementation requires the following:

- ❑ Implement the FRU with a 24C02-compatible SEEPROM, on a private I²C bus of a management controller.
- ❑ Assign a unique Private I²C Bus address of your choice for the FRU. The address depends on the particular microcontroller.
- ❑ Create an SDR for the FRU, using SDR Type 11h (FRU Device Locator).
- ❑ Add the new SDR to the SDR Repository load file, and load it into the Repository.
- ❑ Implement the Master Write-Read I²C command in the appropriate management controller.
- ❑ Design the contents of the FRU Inventory. Refer to *Section14.2: FRU Inventory Area Format.*
- ❑ Create a load file for the FRU Inventory and load it to the device.

14.1.3 IPMI FRU Inventory Directly on the IPMB

These FRU Inventory devices are accessed by the BMC using Master Write-Read Commands.

Implementation requires the following:

- ❑ Implement the FRU Inventory with a 24C02-compatible SEEPROM, directly on the IPMB.
- ❑ Assign a unique slave address for the FRU. Refer to the *IPMB Address Allocation* document. Select one of the "SEEPROM" addresses with "Use" type of "c" (for chassis vendor use) or "b" (for board set vendor use).
- ❑ An FRU Device Locator SDR is required for the FRU in this implementation.
- ❑ Implement the Master Write-Read I²C command in the BMC.
- ❑ Design the contents of FRU Inventory. Refer to *Section14.2: FRU Inventory Area Format.*
- ❑ Create a load file for the FRU Inventory and load it to the device.

14.2 FRU Inventory Area Format

The FRU Inventory data contains information such as the serial number, part number, asset tag, and a short descriptive string for the FRU. The contents of a FRU Inventory area are specified the *Platform Management FRU Information Storage Definition*

14.3 How to Initialize (Program) FRU Devices

Initial programming may be accomplished by a method that suits your manufacturing requirements. For example,

- Access the FRU device using pins on a circuit test fixture.
- Access the device via a connection to the IPMB. Go directly to the device, or through associated management controller.
- Access from System Management Software.
- Obtain parts pre-programmed from parts vendor.

15. How to Implement a Generic Non-Intelligent Device

"Generic" non-intelligent devices are non-intelligent devices other than FRU SEEPROM devices, such as special management ASICs or proprietary controllers, that are on the IPMB or Private Bus and are intended to be accessible via Master Write-Read I^2C command.

In general, use of these devices is NOT recommended, because they will create additional traffic on the IPMB (because the devices are on the IPMB), they are part of the IPMI standard, they may be less reliable (no checksum, for example), and they are NOT supported by cross-platform System Management Software.

Implementing a generic non-intelligent device requires the following:

- Assign a unique slave address for the non-intelligent device. Refer to *IPMB Address Allocation*. You can generally use the "c" (for chassis vendor use), "b" (for board set vendor use) or "a" (for third-party add-ins) addresses.
- If you want System Management Software (SMS) to access the device using Master Write-Read I^2C, implement a Generic Device Locator Record SDR (SDR Type 10h).
- It is recommended that you implement Generic Device Locator Record SDRs for all non-intelligent devices on the IPMB. This is to allow SMS to list devices on the IPMB.
- If the device is not on the IPMB, and you don't provide Master Write-Read access, you don't need an SDR.
- Add the new SDR to the SDR Repository load file, and load it into the SDR Repository.

16. How to Implement Sensors and SDRs

16.1 Overview

This section introduces sensor and Sensor Data Record (SDR) terminology and parameters, then illustrates how to use those parameters to design your sensors.

NOTE

Tables 29-1, 29-2, and *29-3* referenced in this section are from the *IPMI Specification.*

Table 29-1: Event/Reading Type Code Ranges

Table 29-2: Generic Event/Reading Type codes

Table 29-3: Sensor Type Codes

16.2 Introduction to Sensor and SDR Terminology

Sensors are described by their respective SDRs using "code values" representing sensor type, event trigger type, sensor reading type (the types of present reading a sensor provides), and other values. System Management Software uses these codes to recognize, access, configure, interpret, and scan sensors.

This section briefly describes key sensor parameters. For more detailed descriptions, see the *IPMI Specification.*

16.2.1 General Sensor Functions

A sensor can perform one or more of the following:

- Generate Sensor Event Messages (discrete or threshold-based events)
- Return Sensor Readings (analog or discrete readings)
- Return Sensor Event Status

Sensor Event Messages are asynchronously generated messages that carry information about a sensor event. Sensors can generate event messages when their Event Trigger conditions are met, by sending Platform Event (a.k.a. Event Message) commands to the event receiver (nominally the BMC) to be logged in the SEL. A sensor event can be triggered when a specified threshold value is crossed or a discrete event becomes asserted (a state transition).

Sensor Readings are values that sensors return in response to Get Sensor Reading commands. Sensors may scan to obtain the reading after receiving the request, or may return a the presently stored version of a periodically updated reading. A sensor can be designed to return analog or discrete readings.

Sensor Event Status indicates whether an Event Trigger condition has been met and an event message generated by a sensor. Sensor Event Status can be used for event detection by polling in systems where a sensor does not generate event messages or instead of using the System Event Log. Sensors can return sensor event status in response to Get Sensor Event Status commands. The kind of status returned depends on the sensor type, threshold based or discrete.

16.2.2 Sensor Data Records

In general, a sensor is described by an SDR using one of the two sensor SDR Type formats*: Full Sensor Record* (Type 01h) or *Compact Sensor Record* (Type 02h). *Full Sensor Records* can be used to describe any type of sensor. *Compact Sensor Records* save space, but have limitations in the sensors they can describe. A *Full Sensor Record* is a 64-byte record, while the Compact record is 48-bytes.

In general, a *Full Sensor Record* is used if a sensor :

1) Returns threshold-based present readings, or

2) Generates threshold-based events or status, or

3) Requires settable hysteresis

Otherwise, a *Compact Sensor Record* is used. Additionally, *Compact Sensor Records* allow for SDR record sharing. SDR sharing allows an SDR to be used by multiple sensors having the same Sensor Type Code and other characteristics. For example, the SDR for power supply presence sensors may be shared by several such sensors.

SDR Type differences are described in more detail in the *SDR Type 01h, Full Sensor Record* section of the *IPMI Specification.*

NOTE: It is possible to implement a sensor without a corresponding SDR, logical (versus physical) sensors that only generate event messages, such as Sensor type *Critical Interrupt*, which generates events upon detecting actions such as Bus Timeout or Software NMI.

16.2.3 Sensor Reading Types

Sensor readings are the values sensors return in response to Get Sensor Reading commands. Sensor reading types are defined by the *Event/Reading Type Code* field of either a *Full Sensor Record* or *Compact Sensor Record.*

Threshold-based sensors always return 8-bit analog readings. The format of a sensor's reading is described by the *Sensor Units* and related fields in its associated SDR. System Management Software can use these values for converting the reading from the sensor into units such as volts or degrees centigrade.

For discrete sensors, the Event/Reading Type Code *specifies the possible states the sensor provides. The SDR* Reading Mask *field indicates which possible states could be returned from the sensor as a reading.*

Table 16.2-1: Sensor Reading Types

Sensor Reading Type	Description
Discrete (includes Generic, Sensor-specific, and OEM Discrete)	Sensor returns discrete readings. Digital sensors are considered as discrete sensors that return 2 discrete states.
Threshold based	Sensor returns continuous range of values, such as analog sensors.
None (no reading support discrete sensors only	*Sensor does NOT provide present readings,for example, the sensor does not support Get Sensor Reading commands. Applies to a discrete sensor that only generates sensor events.*

NOTE: Sensors are often referred to in terms of their reading types. For example, a *digital sensor* is a discrete sensor returning digital readings (just two states), and an *analog sensor* is a sensor that returns analog readings.

16.2.4 Event Trigger Types

A Trigger Type indicates what conditions trigger sensor event generations, such as the type of threshold crossing or state transitions. Trigger Types are specified by the *Event/Reading Type Code* field of a *Full Sensor Record* or *Compact Sensor Record.*

Some sensors do not generate Event Messages, but only supply sensor readings in response to Get Sensor Reading commands. For these sensors, Trigger Types would be *none*, and this is specified by setting the SDR's *Sensor Capabilities* field, *Sensor Event Message Control Support* bit to "no events from sensor" (11b).

16.2.5 Sensor Trigger Types and Possible Reading Types

The table below summarizes what kind of sensor reading types are available for specific Sensor Trigger Types.

NOTE:

It is possible for a discrete-trigger sensor to have analog present readings. For example, a sensor with *Event /Reading Type Code* of 0Bh may generate events upon "Redundancy Lost" discrete condition , and return sensor present readings of electrical current (analog values).

Table 16.2-2: Event Trigger Types and Possible Reading Types

Event Trigger Type	Description	Possible Sensor Reading Types
Discrete (including Generic, Sensor-specific, and OEM)	Multiple state triggers possible.	Discrete or analog
'digital" Discrete	Discrete with only two states.	'digital" Discrete, analog
Threshold based	Triggers on reading comparison to threshold values. Threshold Offsets may be considered a special case of the discrete enumeration type.	Analog
None*	Sensor does not generate event messages.	Digital, discrete, analog

* Set SDR's Sensor Capabilities field, Sensor Event Message Control Support bit to "no events from sensor.'

16.2.6 Sensor Type Codes

A sensor's Sensor Type is entered into the *Sensor Type* field of a *Full Sensor Record* or *Compact Sensor Record*. Sensor Types include Temperature, Voltage, Fan, Power Supply and other "Sensor Types" listed in *Table 29-3: Sensor Type Codes*.

NOTE:

Check software support before implementing sensors. The System Management Software you use may not support all possible IPMI sensor types.

16.2.7 Event/Reading Type Codes

Event/Reading Type Codes—event "and/or" reading type codes—are used in SDRs and Event Messages.

- **SDRs**
  The code is specified in the *Event/Reading Type Code* field of *Full Sensor Records* and *Compact Sensor Records*. It indicates the sensor's event trigger type and reading type and, in conjunction with the Event Mask bits, what particular conditions may trigger sensor event generation. It also specifies what types of readings the sensor may return. Not all sensors generate events or return readings.

- **Event Messages**
  Event/Reading Type codes indicate the trigger type that generated an event, such as the type of threshold crossing or state transition that produced the event.

General categories of Event/Reading Type Codes are as follows. Refer to *Table 29-1* of the *IPMI Specification.*

Table 16.2-3: Event/Reading Type Code Categories

Sensor Type	Code Range	Description
Threshold based	01h	Its states (offset values) are specified in *Table 29-2*.
Discrete - Generic	02h - 0Bh	Code from this range is further defined in *Table 29-2: Generic Event/Reading Type Codes* with its state (offset) values.*
Discrete – Sensor Specific	6Fh	This code is like a pointer to *Table 29-3.* When code 6Fh is used, a Sensor Type Code from *Table 29-3* is used to select specific sensors, such as Temperature, Voltage or Power Supply sensor, and that the state (offset) value* associated with the selected sensor is used.
Discrete - OEM	70h - 7Fh	Discrete sensors with OEM-defined states
Discrete - Digital	Included in the Discrete range	A digital sensor is treated as Discrete with only two states

*States (offsets) are described in later sections.

For additional details refer to the *Sensor and Event Type Codes* section of the *IPMI Specification.*

16.2.8 States (Offset Values) for Event/Reading Type and Sensor Type Codes

All Event/Reading Type Codes (*Table 29-2*) and most of the Sensor Type Codes (*Table 29-3*) have sets of states (offset values) associated with them. "State" describes possible trigger conditions and reading types for a given sensor type, such as Temperature or Voltage.

A sensor's states (offsets) are defined in two steps, using the SDR's *Sensor Type* and *Event/Reading Type Code* fields.

1. Select the Sensor Type. Select one of the Codes from *Table 29-3*, such as Temperature, Voltage, or Power Supply.

2. Select the *Event/Reading Type Code* from *Table 29-1*. This directly or indirectly determines which table is used for the states (Offset).

   If you want to use the states for "Threshold" from *Table 29-2*, use the Event/Reading Type Code *Threshold* (01h).

   If you want to use the offsets specifically associated with the Sensor Type (*Table 29-3*), use the Event/Reading Type Code *Sensor-specific* (6Fh).

   If you want to use a set of states (offsets) specified in *Table 29-2*, use a code from the Generic range (*Table 29-1*).

   If you want to use OEM-defined Offsets, then use a code from the OEM code range (*Table 29-1*) and define your own offsets for the code. (The definition of OEM specified offsets is controller-specific. Software can use the Get Device ID command to get the Manufacturer ID for the OEM that defined the offsets).

Table 16.2-4: States (Offset) Examples

Sensor Type Code (from Table 29-3)	**Event/Reading Type Code** (which table is used for Offset)	**Event/Reading Class**	**States (Offset)** (from Table 29-2 or 29-3)	**States (Offset Value) Description**	**Comments**
02h (Voltage)	01h (Table 29-2)	Threshold	00h 01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah 0Bh	Lower Non-critical - going low Lower Non-critical - going high Lower Critical - going low Lower Critical - going high Lower Non-recoverable - going low Lower Non-recoverable - going high Upper Non-critical - going low Upper Non-critical - going high Upper Critical - going low Upper Critical - going high Upper Non-recoverable - going low Upper Non-recoverable - going high	This Voltage sensor generates Event Messages triggered by Threshold events using state values to the left. They are from Table 29-2, Code 01h.
09h (Power Unit)	02h (Table 29-2)	Discrete, Generic	00h 01h 02h	Transition to Idle Transition to Active Transition to Busy	This Power Unit sensor generates Event Messages triggered by discrete events using state (Offset) values to the left. They are from Table 29-2, Code 02h.
09h (Power Unit)	6Fh (Table 29-3) See Table 29-1 also.	Discrete, Sensor-specific. 09h from Table 29-3	00h 01h 02h 03h 04h 05h 06h	Power Off / Power Down Power Cycle 240VA Power Down Interlock Power Down A/Clost Soft Power Control Failure Power Unit Failure detected	This Power Unit sensor generates Event Messages triggered by discrete events using state (Offset) values to the left. They are fromTable 29-3, Code 09h (Power Unit). (Event/Reading Code 6Fh indicates that the states (Offset) comes from Table 29-3.)
14h (Button)	03h	'digital" Discrete	00h 01h	State Deasserted State Asserted	This Button sensor generates Event Messages triggered by digital events using state (Offset) values to the left. They are defined in Table 29-2, Code 03h.

16.2.9 Masks on States (Offsets)

For a given sensor, Event Masks (Assertion and Deassertion Event Masks) and a Reading Mask are applied on the same set of States. These Masks can be different.

16.2.10 Event Masks

Event Masks are bit masks on the possible states (Offset values), which determine the sensor's trigger conditions, or what specific conditions will trigger event message generation. The masks are required for all sensors that generate event messages.

Event Masks are specified using the Assertion Event Mask and Deassertion Event Mask fields of an SDR. The following examples illustrate the effect of event mask settings.

EXAMPLES:

a) SDR settings:
Sensor Type Code = 09h (Power Unit)
Event/Reading Type Code = 0Bh (Discrete). This Code has 3 offset values: *Redundancy Regained*, *Redundancy Lost*, *Redundancy Degraded*.
Assertion Event Mask = 0007h (for non-threshold-based sensors).

The Event Mask is applied on the 3 offset values. As a result, this sensor will generate event messages when *Redundancy Degraded* (Offset = 02h) or *Redundancy Lost* (Offset = 01h), or *Redundancy Regained* (Offset = 00h) becomes asserted.

b) SDR settings:
Same as a) except:
Deassertion Event Mask = 0006h (for non-threshold-based sensors).

This sensor will generate event messages when either *Redundancy Degraded* (Offset = 00h) or *Redundancy Lost* (Offset = 01h) becomes *deasserted*.

c) SDR settings:
Same as a) except:
Assertion Event Mask = 0006h (for non-threshold-based sensors).
Deassertion Event Mask = 0006h (for non-threshold-based sensors).

This sensor will generate event messages when either *Redundancy Degraded* (Offset = 00h) or *Redundancy Lost* (Offset = 01h) becomes *asserted* or *deasserted*.

16.2.11 Reading Masks

The Reading Masks are bit masks on the possible offset values associated with sensors. The masks are defined using the *Reading Mask* field of a *Compact Sensor Record,* and the *Reading Mask/Settable Threshold Mask, Readable Threshold Mask* field of a *Full Sensor Record.*.

For Discrete sensors, the Reading Mask provides information on what specific discrete readings (such as which offset values) are returned in response to Get Sensor Reading commands.

Threshold-based sensors (sensors returning analog readings) always return 8-bit readings and do NOT require Reading Type information. Instead, they need Threshold Masks which are described later in this section.

EXAMPLES:

a) SDR settings:
Sensor Type Code = 09h (Power Unit), from *Table 29-3*
Event/Reading Type Code = 0Bh (Discrete), from *Tables 29-1 and 29-2*
Reading Mask = 0007h.

This sensor returns Discrete readings of all available offsets from *Table 29-2* (00h, 01h and 02h, or *Redundancy Regained*, *Redundancy Lost*, *Redundancy Degraded* respectively).

b) SDR settings:
Sensor Type Code = 09h (Power Unit), from *Table 29-3*
Event/Reading Type Code = 6Fh (Discrete, Sensor Specific), from *Table 29-2*
Reading Mask = 0004h. This mask is on the offsets associated with Sensor Type code 09h (*Table 29-3*).

This sensor returns Discrete reading of the *240VA Power Down* status.

16.2.12 Threshold Masks

These masks are used for sensors that return Threshold-based (analog) readings. Masks include the Settable Threshold Mask and Readable Threshold Mask. The Settable Threshold Mask is used both to tell system management software which thresholds are settable, and to tell the init agent which thresholds it should initialize. Refer to *the Reading Mask/Settable Threshold Mask, Readable Threshold Mask* field of *Full Sensor Records*. The Readable Threshold Mask is also available in *Compact Sensor Records,* in the *Reading Mask* field.

16.2.13 Event Trigger Re-arming and Sensor Event Status

Sensors can be re-armed either automatically or manually. This parameter is specified using the Sensor Auto Re-arm Support bit of the Sensor Capabilities field of SDRs.

An auto-re-arm sensor will re-arm itself when the event condition goes away, such as a transition from "above threshold" to "below threshold." The sensor will return the "present" event status for a Get Sensor Event Status command. The value returned for an auto-re-arm sensor is similar to the "present readings" returned for a Get Sensor Reading command, and in many cases the Get Sensor Reading command is used as a substitute for the Get Sensor Event Status command. Refer to the description of the Get Sensor Event Status command in the *IPMI Specification* for more information.

A manual-re-arm sensor will latch an event status until it is manually re-armed with a Re-arm Sensor Events command (or when all of the controller's sensors are re-armed via the Set Event Receiver command). The sensor will return the "latched" event status for a Get Sensor Event Status command, and "present readings" for a Get Sensor Reading command. In other words, a manual-re-arm sensor can return different values for Get Sensor Reading and Get Sensor Event

Status commands. Thus, a manual-re-arm sensor requires implementing both the Get Sensor Event Status and Re-arm Sensor Events commands.

The following table is an extract from the *IPMI Specification.*

Table 16.2-5: Get Sensor Event Status Response (from IPMI Specification)

Sensor Class	Auto- re-arm	Status Returned
Threshold based	Yes	Present threshold comparison event status. This is redundant to the threshold comparison status returned with the "Get Sensor Reading" command if the sensor has no hysteresis. Otherwise, software can derive the event status from the Get Sensor Reading command if it knows the hysteresis value.
	No	Latched threshold comparison status. Since manual re-arm status is "sticky', the status may be different than the comparison status returned with the "Get Sensor Reading" command.
Discrete	Yes	Present event status represented by a bit mask indicating the event conditions that are presently active on the sensor. Note: this is redundant to the status returned with the "Get Sensor Reading" command if there is no hysteresis associated with the sensor.
	No	Latched event status represented by a bit mask indicating the event conditions that have been detected on the sensor. Since manual re-arm status is "sticky', the status may be different than the comparison status returned with the "Get Sensor Reading" command.

16.2.14 Hysteresis Values

A sensor that has threshold-triggered event generation require hysteresis values including positive and negative hysteresis values. They are described in the Positive- and Negative-going Hysteresis value fields of *Full Sensor Records*.

16.2.15 Sensor Reading "Conversion Factors'

Sensors with analog reading type or threshold-based event trigger use "raw" values for their thresholds and for returning their readings. Software that accesses these sensors must know how to convert the raw values to a value in meaningful units. For example, a voltage sensor that returns raw A/D counts must have that reading converted to a value in volts. The "conversion factors" may be provided either in the SDR (*Full Sensor Record* only) or for some sensors may be retrievable from the sensor interface. For additional details, refer to the *IPMI Specification.*

16.3 How to Design and Add Sensors

This section describes how to design a sensor using the *IPMI Specification*, and what commands to implement in the management controller that manages the sensor.

16.3.1 Overview

The following is a high-level description of how to design and add sensors to a management controller. It is assumed that the BMC and SDR Repository exist..

This section already assumes that the sensor's associated management controller has been designed.

1) Implement a logical "sensor device" in the management controller, and assign a LUN. Start with LUN 00. A LUN can support a maximum of 255 sensors (0 - 254d). FFh is reserved. Implement additional LUNs if you need more sensor space.

2) Under the LUN, assign a unique sensor number (from range 0 - 254d) to the new sensor. (FFh is reserved.) [The sensor number is used as a handle for accessing the sensor and identifying

the sensor in SDRs. There's no relationship between the sensor number and the sensor type. For example, sensor number 12" could be an analog temperature sensor on one controller, and a discrete power supply sensor on another.]

3) Select the SDR Type (01h or 02h) for the sensor.

4) Define sensor parameters for the SDR.

5) Implement sensor commands in the management controller as required by your sensor design.

6) Repeat steps 3) - 6) for each new sensor.

7) Create the load file for the SDR Repository.

8) Add all new SDRs for your sensors, management controllers, and FRU devices to the SDR Repository load file.

9) Load the load file into the SDR Repository.

16.3.2 Detailed Step-By-Step Procedure for Sensor Design

This section illustrates how to design SDRs and what IPMI commands to implement for your sensors. All the key parameters of sensor SDRs (full and compact) are discussed.

NOTE

Unless otherwise noted, all tables referenced in this section are from the IPMI Specification.

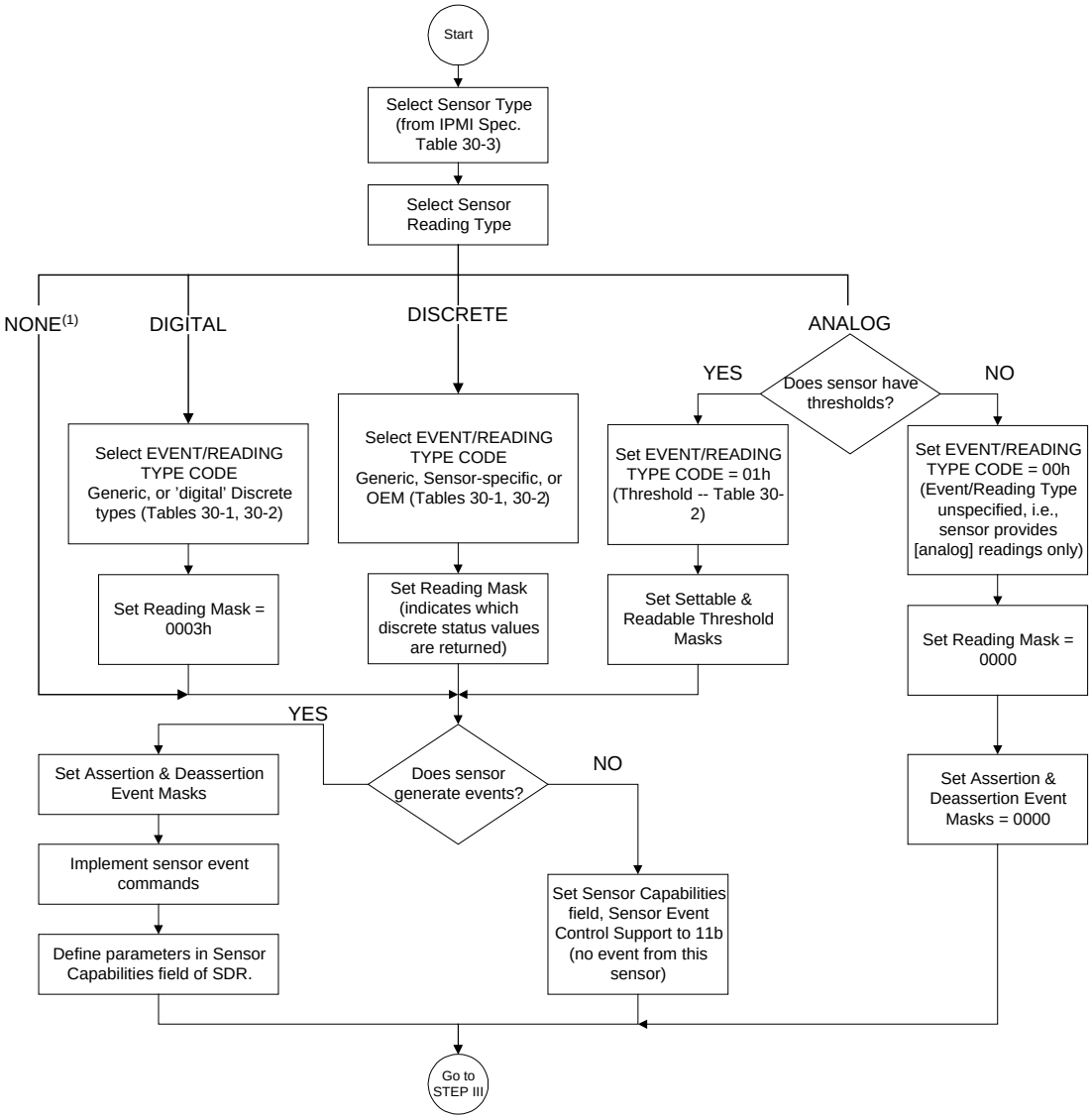
STEP I:

1) Select the SDR Type for your sensor:
Does your sensor (a) return analog readings or (b) require threshold value initialization? Also check against the "difference table" at the top of *the Full Sensor Record* section of the *IPMI Specification.*

If YES, use a *Full Sensor Record*.
If NO, you can use a *Compact Sensor Record*, or if you prefer, you can use a *Full Sensor Record*, unless you would like to implement record sharing.

STEP II:

STEP II describes the process of defining a sensor and associated SDR.



NOTES:
(1) Means sensor does not provide sensor readings, but requires initialization. A “logical” sensor that generates events, but is not accessed, initialized, or controlled via IPMI commands is not required to have an SDR. For example, BIOS may implement a logical sensor for logging POST errors to the SEL.(2) OEM “digital” Event/Reading Types are implemented as a “discrete” with just two states.

Figure 16-1: Sensor Design Flow Chart 1: For All Sensors

1) See if your desired Sensor Type is specified in *Table 29-3: Sensor Type Codes* of the *IPMI Specification.* For example, Temperature, Fan, or Button.

If the desired Sensor Type is in the table, go to next step.

If the desired Sensor Type is not in the table, you need to add an OEM-defined Sensor Type.

2) Determine the Reading Type for your sensor:

- None
  This means that the sensor does not return readings, for example, it does not support the Get Sensor Reading command, but requires initialization or provides event status. This is valid for discrete sensors only. Threshold based sensors must support the Get Sensor Reading command. Set the Reading Mask field of the SDR to 0. Go to step 7.

- Digital or discrete
  Go to step 3 (digital) or step 4 (discrete)

- Analog
  Go to step 6

3) If DIGITAL:

Select the set of Offset values you want to use. To do this, select an Event/Reading Type Code from the Generic range (02h-0Bh) from *Table 29-1*. Go to *Table 29-2* and select a code with "digital" Discrete Offset, such as 03h, 08h. Set the Reading Mask to 0003h. A digital reading sensor always has a Reading Mask of 0003h. When done, go to step 7.

NOTE:
'Digital" sensors cannot use Event/Reading Type Code 6Fh (Sensor-specific), or OEM defined codes (70h-7Fh) in *Table 29-1*; these codes are available for Discrete sensors only. (Digital sensors may be declared as Discrete with just two states, if necessary)

EXAMPLE: Sensor for a power supply
Sensor Type = 08h (Power Supply); see step (1).
Sensor Reading Type = Digital
Event/Reading Type = 05h ('digital" Discrete) from *Table 29-2.* This code has two Offset values: 00h (Limit Not Exceeded) and 01h (Limit Exceeded).
Reading Mask = 0003h. (The MSBs of the Mask are always 0. A mask of 0003h corresponds to bits 0 and 1 being set, indicating the sensor will return status for both the Limit Not Exceeded and Limit Exceeded states, respectively.)

The SDR for this sensor will have the following values:
'Sensor Type" field = 08h (Power Supply)
'Event/Reading Base Type" field = 05h ('digital" Discrete)
'Reading Masks" field (for non-threshold-based sensors) = 0003h

4) If DISCRETE:

Select the set of Offset values you want to use. To do this, select the appropriate Event/Reading Type Code from *Table 29-1* from the Generic, Sensor-specific, or OEM category. Set the Reading Mask according to the states your sensor will actually return. This will typically be a subset of the possible states for the Event/Reading Type Code you selected. When done, go to step 7.

[In actuality, Event/Reading Codes for "digital" sensors are just a subset of the Codes in the Generic Discrete sensor category (the preceding steps for "digital" are given as a shorthand approach - you could follow the Discrete sensor steps and get to the same result). The Event/Reading Type Code for Threshold based sensors is taken from the Generic Discrete sensor category, too.]

Which Code category do you want to use?

4-a) If "Sensor Specific"

It's typical to first check to see if the sensor-specific states for a given sensor type meet the

needs of your sensor. (Remember, even though a sensor type has sensor-specific states defined, you're not obligated to use them. You can use a Generic (or OEM) Event/Reading Type Code with any sensor type.)

Go to *Table29-3* to determine whether the Sensor Type you selected has Sensor-specific Offsets. If the Sensor-specific Offset column has a dash "-', it has no Offsets defined.

Set the Event/Reading Type Code of 6Fh *(Table 29-1)* to indicate your sensor returns Sensor-specific Offsets (states).

Does your Sensor Type have Sensor-specific Offsets?
If NO, you cannot use this method for selecting the Event/Reading Type Code. Try option 5-b) or 5-c) below.

If YES, you have the option of using the Offset values for your Sensor Type, in conjunction with 6Fh from *Table 29-1*.

EXAMPLE: Sensor for a power supply
Sensor Type = 08h (Power Supply); see step (1).
Sensor Type 08h has Offsets. Available Offset values are 00h, 01h or 02h
Sensor Reading Type = Discrete; All Offsets in *Table 29-3*) are Discrete.
Event/Reading Type = 6Fh from Table 17-1
Reading Masks = 0006h. This selects Failure Detected (01h) and Predictive Failure Asserted (02h).

The SDR for this sensor will have the following values:
'Sensor Type" field = 08h
'Event/Reading Type" field = 6Fh
'Reading Masks" field (for non-threshold based sensors) = 0006h

4-b) If "Generic':
Go to *Table 29-2*, and select a code with Discrete Offset. For example, Codes 02h, 07h, 0Ah. (You can use Generic Offsets for any sensor type—regardless of whether that sensor type has Sensor-specific Offsets defined, or is an OEM sensor type.)

EXAMPLE: Sensor for a power supply
Sensor Type = 08h (Power Supply); see step (1).
Reading Type = Discrete; see step (2).
Event/Reading Type = 0Ah (Discrete), from *Table 29-2;* this code has nine Offset (state) values: 00h through 08h.
Reading Mask = 001Dh. This corresponds to Offset values of 00h, 02h, 03, and 04h (transition to running, transition to Power Off, transition to On Line, and transition to Off Line, respectively).

This sensor will return Discrete readings on the specific status defined by the Reading Mask.

The SDR for this sensor will have the following values:
'Sensor Type" field = 08h (Power Supply)
'Event/Reading Type" field = 0Ah (Discrete)
'Reading Mask" field (for non-threshold based sensors) =001Dh

4-c) If "OEM" defined Event/Reading Type Codes
You have the option of returning OEM-defined Offsets - even with an IPMI defined sensor type. You can, of course, also use OEM-defined Offsets with an OEM sensor type.

Sensors with OEM-defined Event/Reading Type Codes are treated as Discrete sensors with OEM-defined offsets. Digital types are implemented as Discrete with only two states.

5) If ANALOG:
Does sensor have thresholds?

If YES,
Set Event/Reading Type Code = 01h (Threshold) (*Tables 29-1 and 29-2*). Set the Settable Threshold Mask and Readable Threshold Mask, which use the same SDR field as the Reading Mask. This indicates the thresholds for which status is returned, and indicates which thresholds can be set. The settable threshold field also tells the init agent which thresholds to initialize if the "Init Thresholds" bit is set.
EXAMPLE: Sensor for a power supply
Sensor Type = 08h (Power Supply); see step (1).
Sensor Reading Type = Analog
Settable Threshold Mask (for threshold-based sensors) = 22h (upper and lower critical thresholds are settable and can be initialized if "Init Thresholds" bit is SDR is set. Refer to the *Full Sensor Record* section of the *IPMI Specification*.)
Readable Threshold Mask (for threshold-based sensors) = 33h (the status of upper and lower non-recoverable thresholds and upper and lower critical thresholds are readable)

The SDR for this sensor will have the following values:
'Sensor Type" field = 08h (Power Supply)
'Event/Reading Type" field = 01h (Threshold)
'Settable Threshold Mask" = 22h
'Readable Threshold Mask" = 33h

If NO, set the SDR as follows:
'Sensor Type" field = 08h (Power Supply)
'Event/Reading Type" field = 00h ('unspecified," meaning sensor provides analog readings only.)
'Reading Mask" field (for non-threshold-based sensors) = 0000h
Go to step III (DONE).

6) Does sensor generate events?
If NO, set the SDR's *Sensor Capabilities* field, *Sensor Event Message Control Support* bits to 11b (no events from sensor). Also set the *Assertion Event Mask* and *Deassertion Event Mask* to 0000h. Go to step III (DONE).

If YES,
a) Set the Event Trigger Masks: *Assertion Event Mask* and *Deassertion Event Mask*. These are bit masks on the sensor's Offset values, and describe which specific Offset conditions your sensor will trigger event messages or return event status on. The *Assertion Event Mask* specifies which conditions (Offset values) will trigger event generations when the conditions become asserted. The *Deassertion Event Mask* specifies which Offset conditions will trigger event generations when the conditions become deasserted.

EXAMPLE: Generic Sensor for a power supply:
Sensor Type = 08h (Power Supply), from *Table 29-3*.
Event/Reading Type = 0Ah (Discrete), from *Table 29-2*.
Reading Mask = 01FFh (example—for all Offset values)

Assertion Event Mask = 001Dh
Deassertion Event Mask = 0002h

Event/Reading Type Code 0Ah has nine possible Offset values, 00 through 08h. *Event Trigger Mask* of 001Dh indicates that Offset values of 00h, 02h, 03, and 04h (*Transition to running, Transition to Power Off, Transition to On Line,* and *Transition to Off Line,* respectively) are selected. Sensor event messages for this sensor will be triggered when any of the above 4 masked conditions become asserted. In addition, the *Deassertion Event Mask* of 0002h indicates that event status will be set on the *deassertion* of Offset 02h, *Transition to Power Off*. This means event status will be set when there is a transition to *Power On.*

The Event Masks and Reading Mask can have the same or different values. The above example showed a sensor that can provide more "present status" readings for more states than it can generated events for - and that it generated events for more assertion events than deassertion events.

b) Implement the following commands:
Set Event Receiver
Get Event Receiver
Platform Event (a.k.a. "Event Message')

c) Set the *Sensor Event Message Control Support* bits of *Sensor Capabilities* field of SDR to 00, 01 or 10, according to your requirements. These bits control whether the sensor generates Event Messages, and if so, what type of Event Message control if offered.

Sensors can be enabled/disabled or controlled in different ways—globally, entire sensor only, or per threshold. "Global" means the only way to enable or disable event generation from the sensor is to enable/disable event generation for the entire management controller. "Per threshold/discrete-state event" means each threshold or discrete-state event (Offset value) within a sensor may be enabled or disabled individually. "Entire sensor only" means that you can only enable/disable event generation for the entire sensor—not for individual states. Refer to the *Full Sensor Record* and *Compact Sensor Record s*ections of the *IPMI Specification.*

If you select "entire sensor only" or "per threshold/discrete-state event," implement the following commands:
Set Sensor Event Message Enable
Get Sensor Event Message Enable

If "global," these commands are not needed, unless you need or want to allow scanning to be enabled/disabled for the sensor by system software.

e) Define whether sensor re-arming is automatic or manual, by indicating the selection with the *Sensor Auto Re-arm* bit in the *Sensor Capabilities* field of SDR.
[You typically use "manual-re-arm" for sensors where it's important to retain the "history" of an event. For example, if you were monitoring current to determine whether a power subsystem was adequately supplying redundant power, you'd use a manual-re-arm sensor to retain whether the system had temporarily drawn more current than could be supplied in a redundant mode.]

If your selection is "manual," implement the following commands:

Get Sensor Event Status – Using this command is the only way to read latched status of sensors requiring manual re-arming.
Re-arm Sensor Events – This command clears the latch and re-enable event generation.

7) DONE. Go to STEP III.

STEP III:

Do you require the BMC to activate your sensor scanning? In other words, will your sensor come up collecting readings by default, or does reading need to be activated?

If YES, initialization is required.
Set the *Init Scanning* bit of SDR's *Sensor Initialization* field to 1. Implement the Set Sensor Event Enable and Get Sensor Event Enable commands for the sensor.

If NO, initialization is not required.
Set the *Init Scanning* bit of SDR's *Sensor Initialization* field to 0.

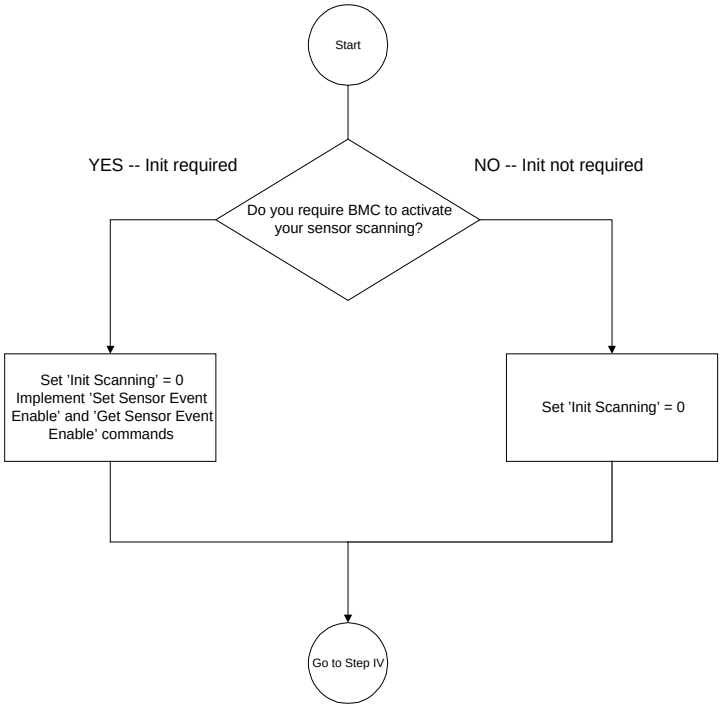


Figure 16-2: Sensor Design Flow Chart 2: For All Sensors

STEP IV:

Does your sensor have thresholds (SDR must be Type 01h)?

If NO, go to STEP V.

If YES, follow the steps below:

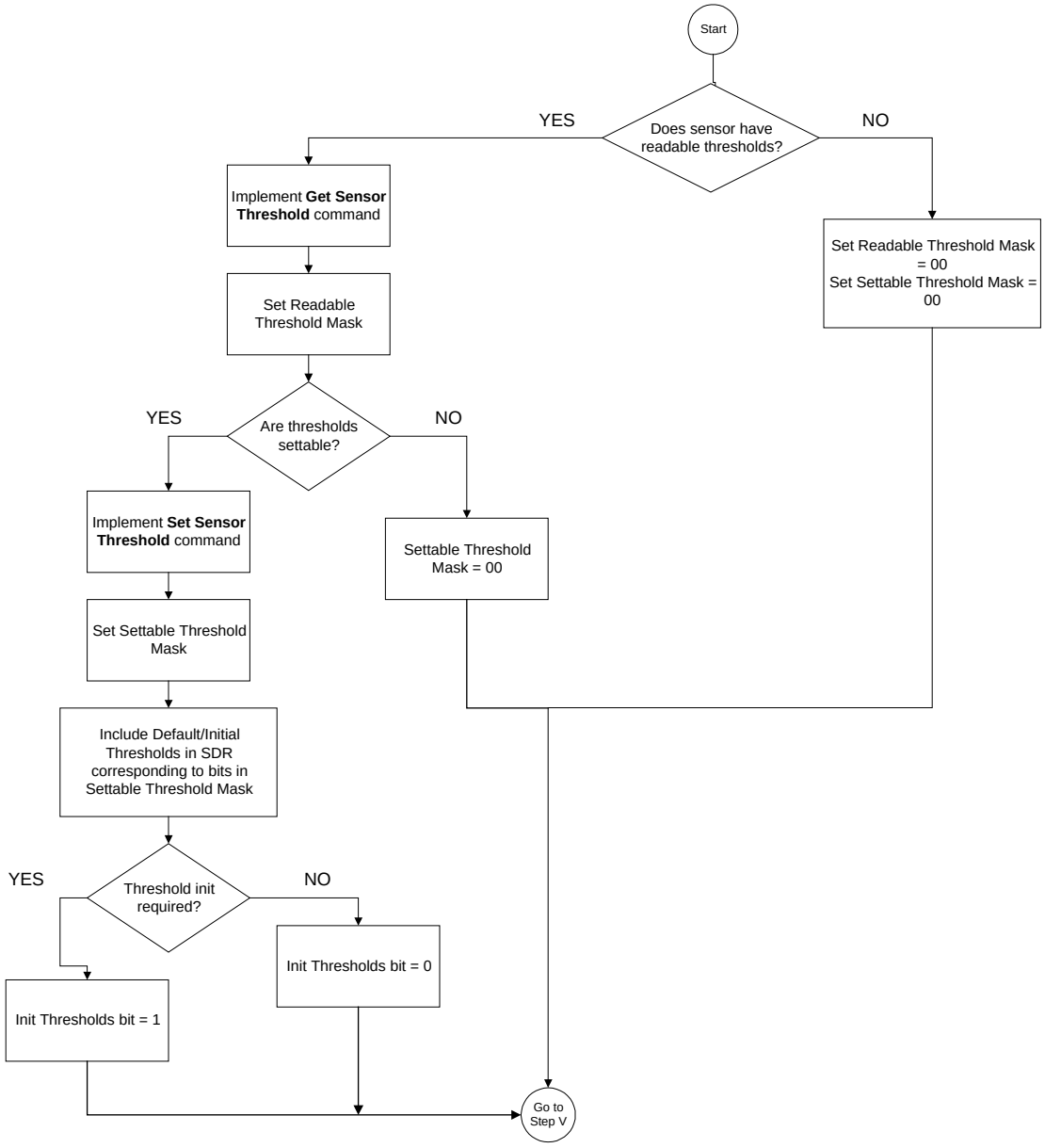


Figure 16-3: Sensor Design Flow Chart 3: For Sensors with Thresholds Only

1) Are thresholds readable? "Readable" means that the sensor returns its specified threshold value in response to commands.

   If YES,

   a) Implement the Get Sensor Threshold command.

   b) Set the *Readable Threshold Mask* bits of the *Reading Mask/Settable Threshold Mask, Readable Threshold Mask* field of the SDR to indicate which thresholds are readable.

If NO, Set both the *Readable* and *Settable Threshold Mask* bits of the *Reading Mask/Settable Threshold Mask, Readable Threshold Mask* field to 0. Go to step 3) DONE.

2) Are thresholds settable? "Settable" means that the thresholds are "writable" via commands. Settable thresholds are required if you want the thresholds to be initialized by the init agent.

If YES,
a) Implement the Set Sensor Threshold command.

b) Set the Settable Threshold Mask bits of the Reading Mask/Settable Threshold Mask, Readable Threshold Mask" field of the SDR to indicate which thresholds are settable.

c) Set default/initial thresholds in SDR corresponding to bits in Settable Threshold Mask. To do this, set the Settable Threshold Mask bits of the "Reading Mask/Settable Threshold Mask, Readable Threshold Mask" field of the SDR to indicate which specific thresholds are settable. You can make these thresholds settable: Upper non-recoverable, Upper critical, Upper non-critical, Lower non-recoverable, Lower critical and Lower non-critical thresholds. Refer to the *Full Sensor Record* section of the *IPMI Specification* for additional instructions.

3) Does your sensor require threshold initialization? If YES, set the *Init Thresholds* bit of the *Sensor Initialization* field of the SDR to 1. If NO, set the bit to 0.

4) DONE. Go to STEP V.

STEP V:

Set other parameters in SDR.

1) For Full Sensor Records
Full Sensor Records are typically used for sensors with thresholds, or sensors that provide analog readings.

a) **Is Sensor Hysteresis settable?** Refer to *Full Sensor Record, Sensor Capabilities* field in the *IPMI Specification*.

Set the *Sensor Hysteresis Support* bits of the *Sensor Capabilities* field to 10b (hysteresis is readable and settable). Implement Get Sensor Hysteresis and Set Sensor Hysteresis commands. ('Not settable" means that hysteresis is hard-coded.)

b) **Linear or Non-Linear?** Is it a Linear/Linearizable sensor or Non-Linear sensor? If Non-Linear, implement Get Sensor Reading Factors command, which is required for polling non-linear sensors.

NOTE: Check your system software's support prior to implementing non-linear sensors.

2) For Compact Sensor Records
This type is typically used for digital/discrete sensors that do not provide settable thresholds or require threshold initialization. Sensors with analog readings or that require settable thresholds must use the *Full Sensor Record.*

a) Sensor Record sharing: The Compact Sensor Records allow multiple sensors of the same type (and that return the same states) to share a single sensor record. For example, you could

use the use on Compact Sensor Record to describe a set of memory presence sensors. If the same SDR is to be shared by multiple sensors of the same type, set parameters in the Sensor Record Sharing field of the SDR. Note that the sensor numbers must be sequential for all sensors sharing a given Compact Sensor Record.

3) For all sensors:
a) Define other parameters required by the particular SDR.

16.4 How to Add OEM-defined Sensors

OEM-defined sensors must be of Discrete Event/Reading Type with Discrete or "digital" Discrete Offsets.

NOTE:

Full use of OEM-defined sensor types typically requires special support in the System Management Software. Depending on how the sensor is implemented, cross-platform System Management Software may be able to generically handle some portion of the sensor.

There are three implementation methods:

- **OEM Sensor Type with Generic (IPMI defined) Offsets.** Use a sensor type code from the "OEM RESERVED" range (C0h - FFh) from Table 29-3. Use an Event/Reading Type from the Generic category. You can use digital, discrete, or threshold. Generic System Management Software wouldn't know what type the sensor is, but it would be able to provide some interpretation of readings and events from the sensor.

- **Standard (IPMI defined) Sensor Type with OEM-defined Offsets.** Use a pre-defined sensor type code from Table 29-3 (such as voltage). Use an Event/Reading Type Code from the OEM code range (70h - 7Fh, Table 29-1). This identifies the offsets as OEM "sensor-specific" offsets. The meaning of the offsets is provided by the manufacturer identified by the Manufacturer ID for the satellite management controller under which the sensor resides. Refer to the Sensor and Event Code Tables section of the *IPMI Specification.* Generic System Management Software would know the sensor type, but wouldn't know how to interpret the Discrete readings or events from the sensor (with the exception that an Event Message supports an optional standard "severity" field - that could be provided).

- **OEM Sensor Type with OEM-defined Offsets.** Use a sensor type code from the "OEM RESERVED" range (C0h - FFh) from Table 29-3 and an Event/Reading Type Code from the OEM code range (70h - 7Fh, Table 29-1). This defines a "totally OEM" sensor, where Generic System Management Software cannot interpret the sensor type, or discrete readings or events from the sensor (with the exception that an Event Message supports an optional standard "severity" field - that could be provided).

1) Can you use one of the "Sensor Types" in Table 29-3?
If YES, go to next step.
If NO, implement a new Sensor Type using one of the OEM RESERVED Codes (C0h - FFh) from *Table 29-3.*

EXAMPLE:
Sensor Type = C0h (Define this code as Acoustic Cancellation sensor, for example.)

2) Can you use one of the Discrete Event/Reading Type Codes from *Table 29-2?*
If YES, use it for your sensor.

EXAMPLE:
Sensor Type = C0h (OEM Acoustic Cancellation Sensor), defined in step 1)
Sensor Reading Type = Discrete
Event/Reading Type = 0Bh (Discrete) from Table 29-2. This Code has three Offset Values, 00h, 01h, and 02h (Redundancy Regained, Redundancy Lost, Redundancy Degraded, respectively.)
Reading Mask = mask on Offsets for Event/Reading Type Code 0Bh
Event Trigger Mask = same as Reading Mask or can be different

If NO, you need to define a new set of Offset values, in one of the two ways:
2-a) Implement one of the Codes from the OEM (70h - 7Fh) from *Table 29-1* and define a set of Offset values for it.

EXAMPLE:
Sensor Type = 13h (Critical Interrupt) from *Table 29-3*
Sensor Reading Type = Discrete
Event/Reading Type = 70h (Discrete - OEM specific); refer to *Table 29-1,* OEM Code category.
Define Offset values for the new code 70h as: 00h, 01h, 02h, and 03h (OEM INTR 0, OEM INTR 1, OEM INTR 2, OEM INTR 3)
Reading Mask = mask on Offsets for new Event/Reading Type of E8h
Event Trigger Mask = same as Reading Mask or can be different

2-b) Implement Offset values for the new Sensor Type Code C0h (you defined in step 1; ref. Table 29-3.) Select 6Fh from *Table 29-1,* so the Offset from Table 29-3 can be used.

EXAMPLE:
Sensor Type = C0h (Torque Sensor), defined in step 1)
Sensor Reading Type = Discrete
Define Offsets for C0h as: 00h, 01h, 02h, and 03h (Performance above spec., Performance meets spec., Performance lags spec., Install error)
Event/Reading Type = 6Fh (Sensor-specific, Discrete), from Table 29-1
Reading Mask = mask on Offsets for Sensor Type C0h (Torque Sensor)
Event Trigger Mask = same as Reading Mask or can be different

3) If you can't use a pre-defined sensor type and can't use on of the Generic Event/Reading Type Codes, you need to both define a new Sensor Type using one of the OEM RESERVED Codes (C0h - FFh) from *Table 29-3*, and select one of the Event/Reading Type Codes from the OEM (70h - 7Fh) range in *Table 29-1* and define a set of Offset values for it.

EXAMPLE:
Sensor Type = C0h (OEM Acoustic Cancellation Sensor)
Sensor Reading Type = Discrete
Event/Reading Type = 70h (Discrete - OEM-specific); refer to *Table 29-1,* OEM Code category.
Define Offset values for the new code 70h as: 00h, 01h, 02h, and 03h (OEM INTR 0, OEM INTR 1, OEM INTR 2, OEM INTR 3)
Reading Mask = mask on Offsets for new Event/Reading Type of E8h
Event Trigger Mask = same as Reading Mask or can be different

4) Determine other sensor parameters as needed.

16.5 How to Change Sensor Parameters

To Change Temporarily – You can change certain parameters of a particular sensor without changing their default values in the SDR. This is done by using commands such as Set Sensor Hysteresis, Set Sensor Threshold, and Set Sensor Event Message Enable.

16.6 To Change Sensor Default (initialization) Settings

To change the sensor default settings to another default settings, create a new SDR load file incorporating the changes, and load the file into the SDR Repository. (This only applies to sensors that are initialized using their SDRs, that is, sensors with the appropriate bits set in the Sensor Initialization field of the SDRs.

17. How to Implement System Configuration and Initialization

This section describes configuration and initialization functions that an implementer is expected to provide.

17.1 Power-Up "Factory" Configuration Information

The Power-up Factory Configuration information is a set of parameters that the BMC uses as soon as it powers up and initializes itself. The parameters include processor type and status, a pointer to the SDR area, fan type, and last power state.

These parameters are typically stored in the "internal use" area of the FRU Inventory area for the baseboard, or the board on which the BMC resides. Changing these parameters may require a corresponding BMC firmware update. The parameters are analogous to the "CMOS Factory Defaults" that are incorporated in the BIOS code, or the "0-ohm" straps used to alter a baseboard configuration. The location and format of this information within the "internal use" area varies by specific BMC implementation.

17.2 Power-Up Configuration Defaults

Some sensors may power up already enabled. (It is not required that all sensors be initialized by the BMC Init Agent.) Sensors may have optional default values implemented within themselves, for event enables, threshold, and hysterisis. The purpose of the default data is to allow a sensor to detect errors and generate event messages even if the BMC Init Agent has not yet run, or if the SDR Repository does not contain an SDR for that sensor. Sensors that detect "cabinet cover open" or "reset button pressed" conditions fall into this category. You may want the sensor to report unusual conditions before the sensor parameters are initialized by the Init Agent. When the Init Agent runs, the default data can be overwritten by the data from the corresponding SDR record if "Initialization bits" are set in the SDR.

17.3 Setup Configuration Information

The SDRs hold most of the initialization information for the BMC and sensors. The information includes the initialization settings of sensor thresholds, as well as "manufacturing default" values that are used prior to loading the initialization thresholds from the SDRs. The Setup Configuration parameters are always present, regardless of whether the operating system or the System Management Software is loaded. Implementers can change these parameters using utilities they provide.

17.4 Run-time Configuration

The System Management Software can keep its own set of temporary configuration parameters, and can change sensor parameters during run-time. (Note: The SDRs in the SDR Repository do *not* change.) It can also provide an option where these parameters can be restored using the "Setup" values or initialization parameters.

18. TOOLS AND UTILITIES

This section describes recommended tools and utilities for development.

18.1 IPMI Development Tools

- An ICE (In-Circuit Emulator) for your microcontroller chip. The availability of a good emulator is a factor in selecting the microcontroller hardware.
- A logic analyzer and an oscilloscope
- I^2C development and test tools. For example, tools made by Micro Computer Control Corporation, Calibre, Ltd., and others.
- Firmware development tools, such as assemblers, compilers, and debuggers.
- Check the IPMI Web site for available tools.

18.2 Tools Available on the Web

Check the IPMI Web site for available tools, at
http://developer.intel.com/design/servers/ipmi/index.htm.

Currently available tools include:

cmdtool - This tools is run from DOS. It executes an IPMI command entered in hex code, and displays the corresponding Response Data in hex. A sequence of IPMI commands may be executed by using a batch file consisting of multiple cmdtool command lines. Help information is displayed when you type: cmdtool <RETURN>

18.3 Supporting Utilities

This section describes types of utilities you may need.

18.3.1 General Recommendations

Utilities may use mandatory and optional IPMI commands. Optional commands may be used to optimize utility operation, but we recommend utilities be able to function with just mandatory commands, as this aids cross-platform compatibility of utilities.

18.3.2 SDR Creation Utility

This utility supports the process of creating and editing individual SDR files for use with the SDR Repository.

The following are some of the useful features for a utility of this kind:

- The ability to create and edit SDR file down to the bit level of each field in each SDR records.
- The ability to add, modify and delete individual SDR records from a file that contains all SDR records for the system.
- The automatic checking of certain SDR values to see if they are valid.

18.3.3 FRU Creation Utility

This utility supports the process of creating and editing FRU files for FRU Inventory area.

The following are some useful features for a utility of this kind:

- The ability to create and edit each FRU file down to the byte or field level
- The ability to add, modify and delete individual FRU fields and FRU areas.
- The automatic checking of certain FRU values to see if they are valid.

18.3.4 SDR Load and Viewer Utility

This utility supports the process of loading SDR files into the SDR Repository, along with the ability to display the contents of the SDR Repository once it has been programmed. This is typically a DOS real mode utility in order to avoid any protected mode hardware restrictions.

The following are some useful features for a utility of this kind:

- The ability to recognize if the file is a valid SDR load file
- The ability for the user to interactively select which SDR records actually get loaded
- The ability to display the SDR data in a clear, understandable format
- The saving of previously programmed SDR records into an SDR file

18.3.5 FRU Load and Viewer Utility

This utility supports the process of loading FRU files into each FRU Inventory area, along with the ability to display the contents of that area once it has been programmed. This is typically a DOS real mode utility in order to avoid any protected mode hardware restrictions.

The following are some useful features for a utility of this kind:

- The ability to recognize if the file is a valid FRU file
- The ability for the user to interactively select which fields values and FRU areas actually get programmed at runtime
- The ability for the user to replace field values at runtime from network files, such as the selecting of unique serial numbers
- The ability to display the FRU data in a clear, understandable format
- The saving of previously programmed FRU Inventory area into an FRU file
- Notifying the user that, when the BMC Internal Use Area has been overwritten, the Sensor Data Records must be reloaded

18.3.6 SEL Viewer Utility

This utility supports the display of the System Event Log, which is useful during development and field support of the target server.

The following are some useful features for a utility of this kind:

- The ability to display, clear and save the SEL content even when the system is unable to boot a high level OS, such as Windows NT. This allows you to use the utility to examine the SEL as part of a post mortem.
- The ability to display the SEL records either in the raw hex format or in the interpreted text form. Ideally, for the interpreted text form, the utility first retrieves the system SDR information and uses that to help interpret the SEL information so that it is as clear and concise as possible.

18.3.7 Firmware Update Utility

Any management controller code must be updatable. The following are some useful features of a utility of this kind:

- The ability to retrieve address information from an updatable hex code file, for either the BMC or satellite management controllers, then program the proper device without requiring user intervention
- The ability to automatically determine the required interface without requiring user intervention
- A command line switch to enable whether the utility verifies the programmed code or not.

19. EXAMPLES OF S/W - F/W COMMUNICATIONS

This section first introduces IPMI message formats, then illustrates how messages (commands) are used for communications with IPM devices, a.k.a. management controllers, and how non-intelligent devices, e.g. FRU SEEPROMs, are accessed.

19.1 Introduction to IPMI Message Formats and Terminology

The format used to deliver an IPMI message is dependent on the transport the message is sent over. For example, the format of a message across the system interface to the BMC is different than the format for the same message sent to the BMC via the IPMB. In a system with an IPMB, there are three different types of communications that are commonly encountered:

a) **System-BMC Communication** – This is the type of communication that is used when system software communicates with the BMC. This can be used to directly access BMC functions, such as using the *Get Device ID* command to get information about the BMC. By sending a *Master Write-Read* I^2C command to the BMC, system software can access non-intelligent devices on private busses "behind" the BMC, or directly on the IPMB. All messages between the BMC and system software are in "system interface" format.

b) **Controller-Controller IPMB Communication** – This type of communication occurs between any two IPM devices on the IPMB. Requests and responses are in IPMB messaging format. This type of communication is commonly seen when a satellite controller sends an Event Message to the BMC. It is also seen as part of System-Satellite Controller communication (see following).

c) **System-Satellite Controller Communication –** This type of communication occurs when system management software uses the BMC as an interface for communicating with a Satellite Management Controller on the IPMB. When system management software wants to send a message to a satellite controller, it encapsulates the IPMB formatted message within a system interface formatted *Send Message* command to the BMC. The BMC strips the IPMB formatted message out of the *Send Message* command and then forwards it to the IPMB. When the satellite controller responds, it sends the response back as an IPMB formatted message to the BMC. The BMC places the message in its Receive Message Queue, and sets flags that indicate to system software that there is data in the queue. System software will, in turn, use a *Get Message* command to the retrieve the response.

   [note: the receive message queue can receive messages from multiple sources - there's no guarantee whether, after sending a request, you'll receive the matching response, or some asynchronous message from another source. Thus, system software is responsible for verifying fields in the receive message buffer in order to match-up requests and responses.]

Additional information and examples of these types of communications are provided below. Refer also to the *BMC-System Messaging Interface* section of the *IPMI Specification*, and the *Intelligent Platform Management Bus Communications Protocol Specification*..

19.1.1 System—BMC Messaging Formats

These formats are used for communicating between the System Management Software (SMS) and the BMC, and to access non-intelligent devices "behind" the BMC, including any private bus, and directly on the IPMB. The example format shown below is the same as the SMIC messaging

format in the *SMIC Interface* section of the *IPMI Specification*. Refer to the *IPMI Specification* for the format of messages that use the KCS or BT system interfaces.

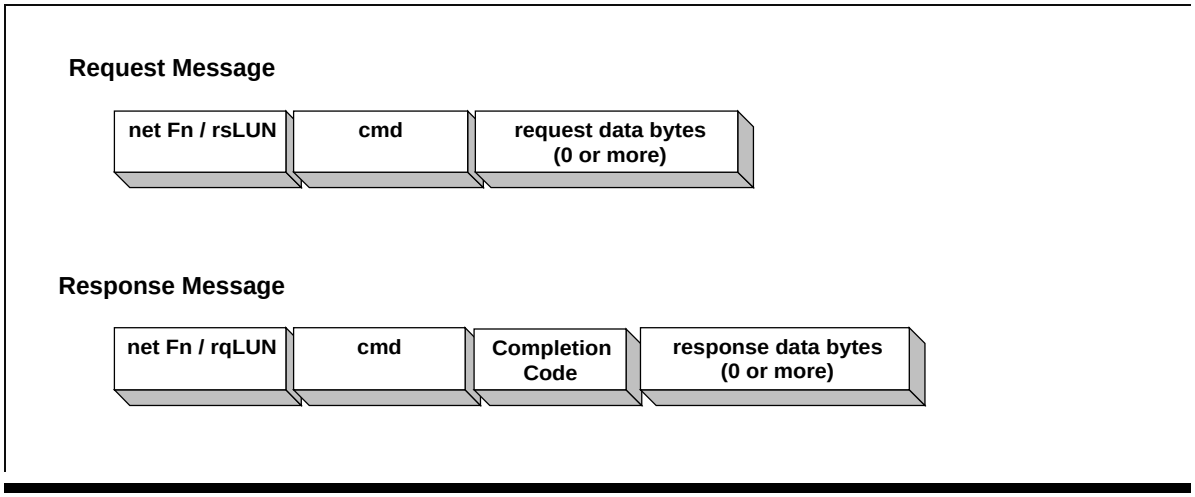


Figure 19-1: System - BMC Messaging Formats (SMIC format shown)

19.1.2 IPMB Messaging Formats

The IPMB Message Formats are used to communicate between IPMB devices on the IPMB. An example is communications between an emergency management card on the IPMB and the BMC, or the transmission of an Event Request Message from a satellite controller to the BMC, and the corresponding Event Response Message sent from the BMC back to the satellite controller. An IPMB formatted message can be encapsulated in a system—BMC message for System-Satellite Controller communication.

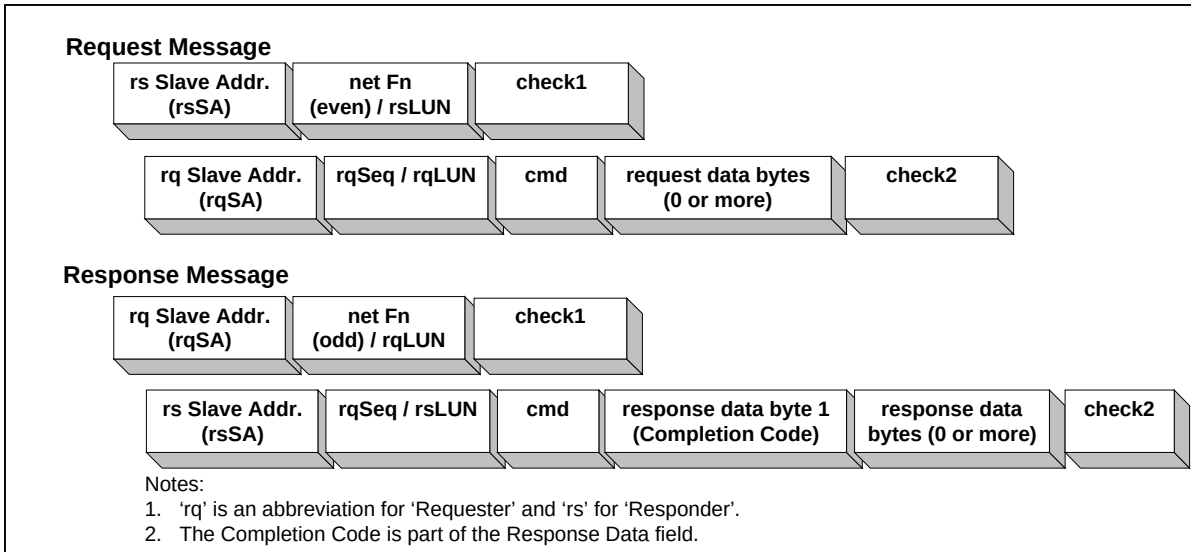


Figure 19-2, IPMB Messaging Formats

Slave Address (SA) is an I^2C address of a device. Specific Slave Addresses are defined in the

Device Record Locator SDRs for the devices. Slave Address allocation information can be found in the *IPMB Address Allocation* document.

Rq is the Requester, or sender of the command.

Rs is the Responder, or receiver of the command.

rqSA is Requester's Slave Address; 1 byte; LS bit is always 0.

rsSA is Responder's Slave Address; 1 byte; LS bit is always 0.

netFn (Network Function) is a 6-bit value that identifies the functional class of a request or response message, such as Application, Storage, Sensor/Event, or Discovery. The NetFn value is even for a request message, odd for a response. For example, a value of 0Ah means a Storage *request*, 0Bh a Storage *response*; 06h and 07h are Application request and response, respectively.

LUN is the Logical Unit Number which occupies the lower 2-bits of the netFn byte; identifies the logical unit number within a management controller, such as BMC, FPC.

rqLUN is Requester's LUN. Identifies the LUN in the Requester device (such as BMC) that the Responder (such as FPC) should send the response to. LUN of 10b identifies SMS Message Buffer within the BMC.

rsLUN is Responder's LUN. Identifies the LUN in the Responder that the Requester should send the command to.

rqSeq is the Requester's Sequence number field. Six bits. This value is picked by the System Management Software and is used as part of verifying that the response matches a particular instance of a request.

Check1 is a two's complement checksum on rsSA and netFn/rsLUN bytes.

Check2 is a two's complement checksum on prior bytes starting with rqSA.

Request data bytes contain information expected by the particular command in a request message. For example, the offset (address) of the first data byte to read or the number of bytes to read.

Response data bytes contain information supplied in a response message in response to a particular command in the corresponding request message. Note that the Completion Code is part of the Response data.

Completion Code is the first data byte in the response message, and indicates whether the corresponding request completed successfully or not. The values for the Completion Code are specified in the *IPMI Specification.*

19.1.3 Send Message, Get Message formats

System-Satellite Controller communications use the Send Message and Get Message commands to transmit messages on the IPMB and to retrieve messages that have been sent to the Receive Message Queue from the IPMB. The Send Message and Get Message commands can be used to transmit and retrieve messages from other interfaces as well. A "channel" number in the command is used to select which interface to transmit a message to, using the Send Message command, and which interface a message was received from, when retrieved from the Receive Message Queue using the Get Message command. Refer to the *BMC-System Messaging Interface* section of the

IPMI Specification.

The IPMB is channel 0, by default. Since a BMC implementation may have other channels and interfaces present, system software must use the channel # to interpret the format of the data returned by the Get Message command. The following figures show the format of the Send Message and Get Message commands to the BMC over a SMIC interface.

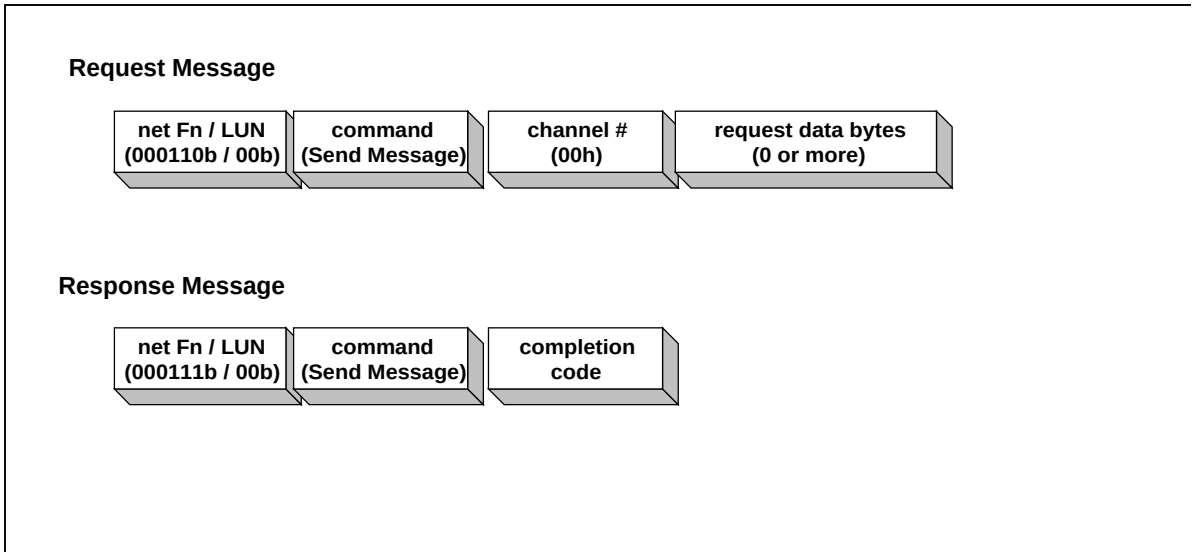


Figure 19-3: System-SMC Messaging Formats for Sending IPMI Commands

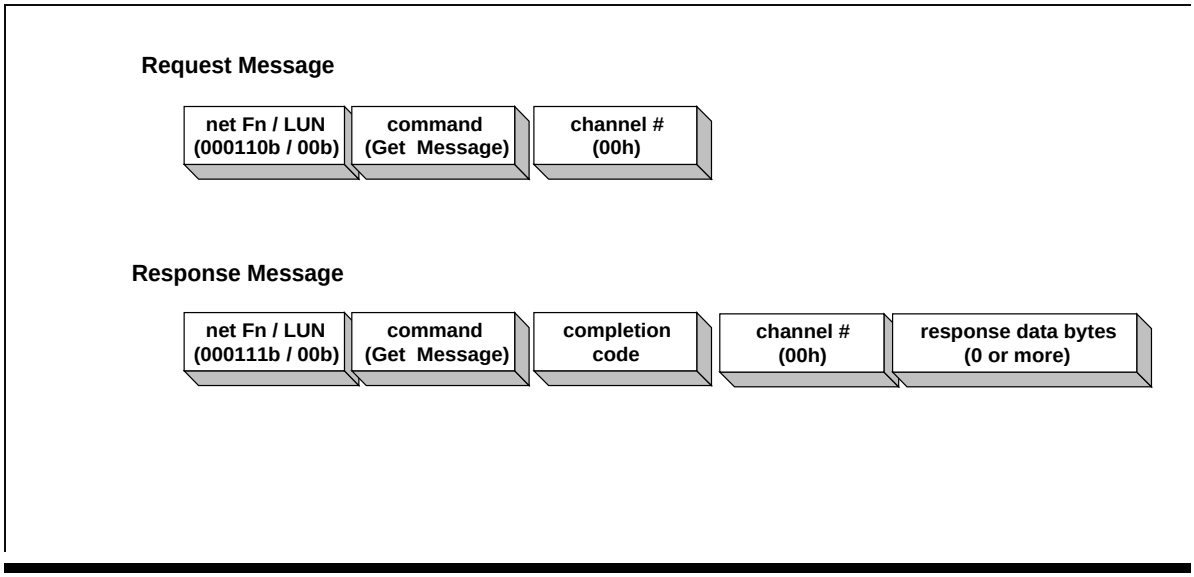


Figure 19-4: System-SMC Messaging Formats for Getting IPMB Response Data

netFn (Network Function) is a 6-bit value that identifies the functional class of a request or response message. The NetFn value is *even* for a request message, *odd* for a response. In the

above figures, the netFn for the Send Message and Get Message commands is "Application," and value 000110b (06h) means an Application *request*, 000111b (07h) an Application *response*.

LUN is the Logical Unit Number that occupies the lower 2-bits of the net Fn byte. In the Request Message, it identifies the logical unit number within the BMC. LUN 00b is used for all messages to the BMC through the System Interface. In the Response Message, it is a return of the LUN that was passed in the Request Message.

Channel 0 is a path through the BMC that allows messages to be sent between the System Interface and IPMB.

Request data bytes contain information expected by the particular command in a request message. These bytes, if present, take the *Request Message* format in *Figure 19-4: System-SMC Messaging Formats for Getting IPMB Response Data.*

Response data bytes contain information supplied in a response message in response to a particular command in the corresponding request message. These bytes, if present, take the *Response Message* format in *Figure 19-4: System-SMC Messaging Formats for Getting IPMB Response Data.*

Completion Code indicates whether the corresponding request completed successfully or not. The values for the Completion Code are specified in the *IPMI Specification.*

19.2 Communication Examples

This section illustrates how System Management Software reads FRU Inventory Information implemented in several different ways, as in FRUs (A) - (E) in the figure below. It is assumed that FRUs and the Satellite Management Controller (SMC) exist as shown.

NOTE:

Message formats in the examples apply to System Interface implemented with SMIC or KCS. The BT implementation will require additional fields. Refer to the *IPMI Specification.*

For additional information on communications, refer to the *BMC-System Messaging Interface* section of the *IPMI Specification*.

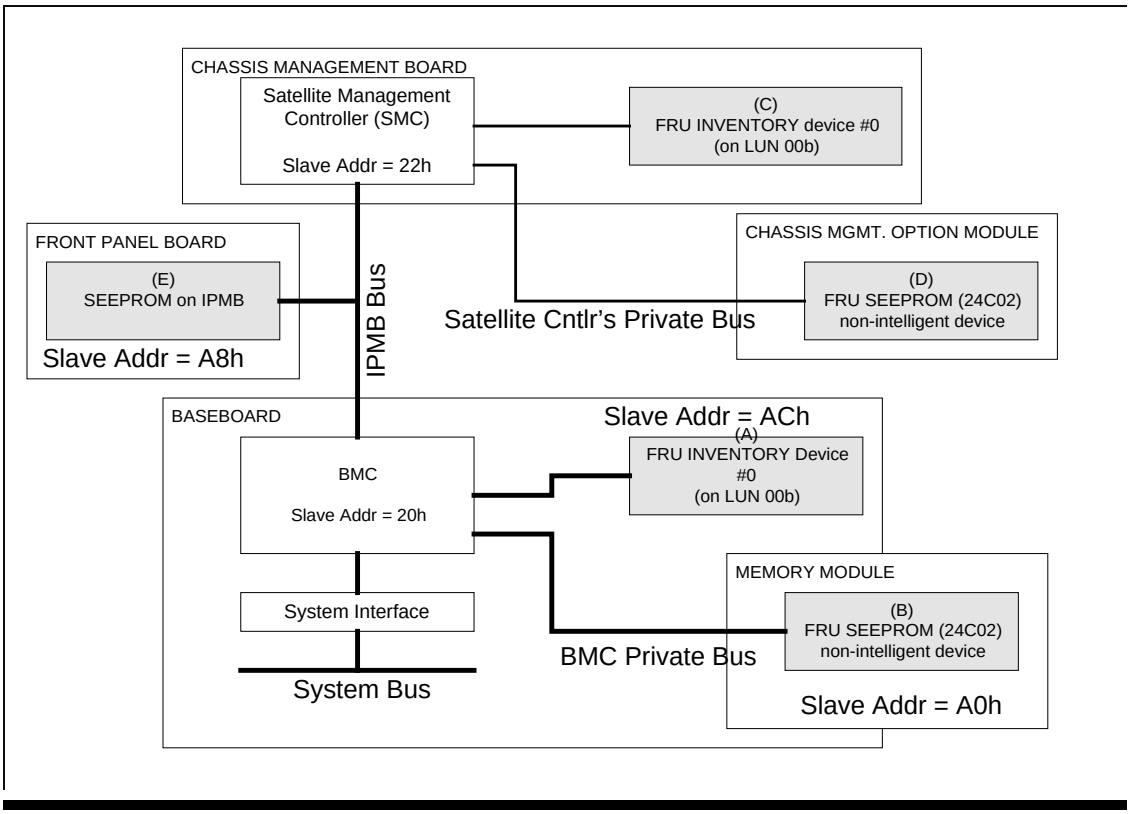


Figure 19-5: Different Implementations of FRUs

19.2.1 Example A-1: Reading FRU implemented "behind" BMC, via IPMB

This example illustrates communications between two management controllers on the IPMB, without involving system software. This type of communication may be seen between an emergency management card and the BMC.

Refer to block (A) of *Figure 19-5: Different Implementations of FRUs.* In this example, assume that the SMC (Satellite Management Controller) is on an emergency management card and wants to access FRU Inventory device #00h at LUN 00b "behind" the BMC [designated as (A) in the figure].

This example shows the format of the messages on the IPMB that read 2 bytes of data beginning at address 0310h of the FRU Inventory device.

Table 19.2-1: Ex A-1 - REQUEST Data for "Read FRU Inventory Data"

Field	byte #	Field Content	Comment
rsSA	1	20h	BMC's Slave Addr
netFn / rsLUN	2	Storage Rq / LUN 00 = 001010b / 00b = 28h	LUN for FRU Inventory associated with BMC
Check1	3	Checksum of two preceding bytes	
rqSA	4	22h	SMC's Slave Addr (requester's Slave Addr.)
rqSeq / RqLUN	5	110101b (example) / 00b = D4h	
Cmd Code	6	Read FRU Inventory Data = 11h	
Data 1	7	FRU Device ID = 00h	00h means this FRU is the primary FRU. This is the FRU that holds the management. controller. For the BMC, this is typically the baseboard FRU.
Data 2	8	FRU Inventory Offset to read, LS byte = 10h	
Data 3	9	FRU Inventory Offset to read, MS byte = 03h	
Data 4	10	Read count = 02h (example; byte count assumed)	Count may be byte or word count depending on what the FRU returns in response to a Get FRU Inventory Area Info command.
Check2	11	Checksum of 7 preceding bytes	

Table 19.2-2: Ex A-1 - RESPONSE Data for "Read FRU Inventory Data"

Field	byte #	Field Content	Comment
rqSA	1	22h	SMC's Slave Addr
netFn / rqLUN	2	Storage Rs / LUN 00 = 001011 / 00 = 2Ch	
Check1	3	Checksum of two previous bytes	
rsSA	4	20h	BMC's Slave Addr
rqSeq / rsLUN	5	110101 / LUN 00 = 110101 / 00 = D4h	
Cmd Code	6	Read FRU Inventory Data = 11h	
Completion Code	7	00h	00h = OK
Data 1	8	Count of data returned = 02h	Byte count assumed
Data 2	9	Data returned from FRU @ offset 0310h = ABh (example)	Byte count assumed
Data 3	10	Data returned from FRU @ offset 0311h = CDh (example)	Byte count assumed
Check2	11	Checksum of 7 previous bytes	

19.2.2 Example A-2: Reading FRU implemented "behind" BMC, via System Interface

This example reads the same FRU as in example A-1, but the FRU is read by the System Management Software (SMS), without involving the IPMB. This is an example of System-BMC communication. Refer to block (A) of *Figure 19-5: Different Implementations of FRUs.*

The following example reads 2 bytes of data from FRU device 00h on LUN 00b, beginning at address 0310h of the FRU Inventory device. Note that the system interface message format does

not contain checksums and slave addresses that are used for messages on the IPMB.

Table 19.2-1: Ex A-2 - REQUEST Data for "Read FRU Inventory Data"

Field	byte #	Field Content	Comment
netFn / rsLUN	1	Storage Rq / LUN 00 = 001010b / 00b = 28h	LUN for FRU Inventory "behind" the BMC
Cmd Code	2	Read FRU Inventory Data = 11h	
Data 1	3	FRU Device ID = 00h	00h means this FRU is the primary FRU of the system, usually the baseboard FRU.
Data 2	4	FRU Inventory Offset to read, LS byte = 10h	
Data 3	5	FRU Inventory Offset to read, MS byte = 03h	
Data 4	6	Read count = 02h (example; byte count assumed)	Count may be byte or word count depending on what the FRU returns in response to a GET FRU Inventory Area Info command.

Table 19.2-2: Ex A-2 - RESPONSE Data for "Read FRU Inventory Data"

Field	byte #	Field Content	Comment
netFn / rqLUN	1	Storage Rs / LUN 00 = 001011 / 00 = 2Ch	BMC LUN00
Cmd Code	2	Read FRU Inventory Data = 11h	
Data 1	3	Completion Code = 00h	00h = OK
Data 2	4	Count of data returned = 02h	Byte count assumed
Data 3	5	Data returned from FRU @ offset 0310h = ABh (example)	Byte count assumed
Data 4	6	Data returned from FRU @ offset 0311h = CDh (example)	Byte count assumed

19.2.3 Example B: Reading FRU on BMC Private I²C Bus, via System Interface

Refer to block (B) of *Figure 19-5: Different Implementations of FRUs.* This example uses the BMC as an I²C controller to access a non-intelligent device on the BMC's Private I²C Bus. The System-BMC Messaging formats are used. It is assumed that the FRU Inventory is implemented with a 24C02-compatible SEEPROM. The Master Write-Read command shown causes the BMC to first write 13h to the SEEPROM device as a master write operation (this selects offset 13h in the SEEPROM). The BMC then issues an I²C "repeated start" and reads two bytes from the SEEPROM using a master read operation.

Table 19.2-3: Ex B - REQUEST Data for "Master Write-Read I²C"

Field	byte #	Field Content	Comment
netFn / rsLUN	1	App Rq / LUN 00b = 000110b / 00b = 18h	BMC LUN 00b
Cmd Code	2	Master Write-Read I²C = 52h	
Data 1	3	Bus ID = 01b	BMC Private Bus #0 (least significant bit = 1 indicates access is to a private bus)
Data 2	4	Slave Address of FRU (in bits 7:1) = A0h (example)	
Data 3	5	Number of bytes to read = 2 (example)	
Data 4	6	Data to write = 13h (example)	Offset within FRU (SEEPROM) to read from

Table 19.2-4: Ex B - RESPONSE Data for "Master Write-Read I²C"

Field	byte #	Field Content	Comment
netFn/rqLUN	1	App Rs / LUN 00b = 000111b / 00b = 1Ch	BMC LUN00
Cmd Code	2	Master Write-Read I²C = 52h	
Data 1	3	Completion Code = 00h	00h = OK
Data 2	4	Data byte read from SEEPROM offset 13h = ABh (example)	
Data 3	5	Data byte read from SEEPROM offset 14h = CDh (example)	

19.2.4 Example C: Reading FRU implemented "behind" Satellite Management Controller, via System Interface

This example illustrates how the System Management Software accesses FRU located behind a Satellite Management Controller (SMC) on the IPMB. Refer to block (C) of *Figure 19-5: Different Implementations of FRUs*. The FRU is implemented as FRU device #00h on LUN 00b of the SMC.

The FRU is accessed using a Read FRU Inventory Data command to the SMC. To get that command to the IPMB (and hence to the satellite controller) System Management Software encapsulates the Read FRU Inventory Data command within a Send Message command and sends the entire command package to the BMC, which then extracts the encapsulated command and puts it on the IPMB to the SMC.

This example reads 2 bytes of data beginning at address 0310h of the FRU Inventory device #00

on LUN 00b. It requires two steps:

Step 1: Send Message

System Management Software encapsulates a Read FRU Inventory Data command (in IPMB format) within a Send Message command to the BMC.

"NetFn" for Send Message commands is "Application"—06h for Request Message, and 07h for Response Message. NetFn for Read FRU Inventory Data is "Storage"—0Ah for request, 0Bh for response.

Table 19.2-5: Ex C - Send Message REQUEST with encapsulated "Read FRU Inventory Data"

Field	byte #	Field Content	Comment
netFn / rsLUN	1	App Rq / rsLUN 00b = 000110b / 00b = 18h	BMC LUN 00b. LUN is always 00b for System interface.
Cmd Code	2	Send Message = 34h	
Channel	3	00h	BMC Messaging Channel 0
Data 1	4	rsSA = Responder's Slave Address (in bits 7:1) = 22h (example)	Beginning of encapsulated message. 22h = Slave Addr to write to on IPMB (slave addr of SMC).
Data 2	5	Net Fn / rsLUN = Storage Rq / LUN00b = 001010b / 00b = 28h	LUN 00 of SMC
Data 3	6	Check 1 = Checksum of two preceding bytes.	
Data 4	7	rqSA =20h	Slave Addr of BMC (requester's Slave Addr., requester on IPMB.)
Data 5	8	RqSeq / RsLUN = Sequence # / SMS Message LUN = 110101b (example) / 10b = D6h	Use of SMS Message LUN (10b) directs the response back to the BMC's Receive Message Queue.
Data 6	9	Cmd Code = Read FRU Inventory Data = 11h	
Data 7		FRU Device ID = 00h (example)	
Data 8	10	Data = FRU Inventory Offset to read, LS Byte = 10h	Beginning of data for Read FRU Inventory Data command
Data 9	11	Data = FRU Inventory Offset to read, MS Byte = 03h	
Data 10	12	Data = Read Count = 02h (byte count assumed)	Count may be byte or word count depending on what the FRU returns in response to a GET FRU Inventory Area Info command.
Data 11	13	Check2 = checksum of 7 preceding bytes	

The following is the response for Send Message.

Table 19.2-6: Ex C - RESP. Data for "Send Message" with encap. "Read FRU Inventory Data"

Field	byte #	Field Content	Comment
netFn / LUN	1	App Rs / LUN 00b = 000111b / 00b = 1Ch	BMC LUN 00b
Cmd Code	2	Send Message = 34h	
Data 1	3	Completion Code = 00h	00h = OK

After receiving the above response to "Send Message," System Management Software waits until the response data from the SMC (such as FRU content) is in the Receive Message Queue, then performs step 2. (When a message is received into the Receive Message Queue, the BMC sets the "ATN" flag in the System Interface.) [System software uses the channel #, Seq field, and other fields to verify that the message it extracts from the Receive Message is the response for the Read FRU Inventory Data request to the SMC.]

Step 2: Get (Response) Message

System Management Software reads the Receive Message Queue to get the response that was delivered to the BMC from the SMC (such as data read from the FRU) by sending a Get Message command to the BMC. This transaction is between the BMC and system side and does not involve the IPMB.

Table 19.2-7: Ex C - REQUEST Data for "Get Message"

Field	byte #	Field Content	Comment
netFn / LUN	1	App Rq / LUN 00b = 000110b / 00b = 18h	BMC LUN 00b. LUN is always 00b when communicating with BMC from system side
Cmd Code	2	Get Message = 33h	

The response from the Receive Message Queue includes the response to the Get Message command followed by the SMC's response to the Read FRU Inventory Data command, as shown in the next table.

Table 19.2-8: Ex C - RESPONSE Data for "Get Message"

Field	byte #	Field Content	Comment
netFn / LUN	1	App Rs / LUN 10b = 000111b / 00b = 1Ch	LUN 00b is always used for System interface.
Cmd Code	2	Get Message = 33h	
Completion Code	3	00h	OK.
Channel #	4	00h	SMS Messaging Channel
Data 1	5	netFN / rqLUN = App Rq **/** SMS Message Buffer LUN = 000111b / 10b = 1Eh	Beginning of data stored in Receive Message Queue. This is the Response Data returned on the IPMB from SMC in response to the Read FRU Inventory Data command sent in step 1 above minus the leading Slave Address (rqSA = BMC) of the Response.*
Data 2	6	Check1 = checksum of the previous byte and BMC Slave Addr (20h) *	BMC Slave Address is included in the checksum calculation although it has been stripped off from the command sequence.*
Data 3		RsSA = 22h	Responder address (SMC addr)
Data 4	7	rqSeq / rsLUN = 110101b / 00b = D4h	00b = SMC Controller LUN. LUN where the message came from.
Data 5	8	Cmd Code = Read FRU Inventory Data = 11h	Response to Read FRU Inventory Data command
Data 6	9	Data = Completion Code = 00h	Completion Code for Read FRU Inventory Data
Data 7	10	Count of data returned = 02h	Byte count assumed
Data 8	11	Data returned from FRU @ offset 0310h = ABh (example)	Byte count assumed
Data 9	12	Data returned from FRU @ offset 0311h = CDh (example)	Byte count assumed
Data 10	13	Check2 = Checksum of 7 preceding bytes	End of Response Data for Read FRU Inventory Data, returned from SMC and stored in SMS Message Buffer in BMC

NOTE:
* The Receive Message Queue holds the entire message received from the IPMB, minus the first byte. This byte gets stripped off when the BMC receives the request. This byte is the BMC's slave address, so the value is already known.

19.2.5 Example D: Reading FRU on Private Bus of Satellite Management Controller, via System Interface

Refer to block (D) of *Figure 19-5: Different Implementations of FRUs.* This example assumes that the FRU Inventory is implemented with a 24C02-compatible SEEPROM on a private management bus.

The example reads 2 bytes of data beginning at address 13h within the FRU Inventory device. System Management Software locates the Slave Address of the FRU Inventory from the information it obtains from the SDR.

In this example, system management software sends a Master Write-Read I^2C command to the SMC by encapsulating it within a Send Message command. Sending the command to the SMC and retrieving the response from the SMC requires two steps:

Step 1: Send Message

System management software encapsulates a Master Write-Read I^2C in a Send Message command to the BMC.

Table 19.2-9: Ex D - "Send Message" REQUEST with encapsulated "Master Write-Read I^2C"

Field	byte #	Field Content	Comment
netFn / rsLUN	1	App Rq / rsLUN 00b = 000110b / 00b = 18h	LUN is always 00b for communicating with System Interface .
Cmd Code	2	Send Message = 34h	
Channel #	3	00h	SMS Messaging Channel
Data 1	4	rsSA = Responder's Slave Address (in bits 7:1) = 22h (example)	Start of IPMB message (Master Write-Read I^2C) to SMC. Slave Addr is that of SMC.*
Data 2	5	= Net Fn / rsLUN = Storage Rq / LUN00b = 001010b / 00b = 28h	LUN 00b = LUN of SMC
Data 3	6	Check 1 = Checksum of two previous bytes	
Data 4	7	RqSA = 20h	Slave Addr of BMC
Data 5	8	RqSeq / RsLUN = Sequence # /SMS Message LUN = 110101b (example) / 10b = D6h	Use of SMS Message LUN directs the response to the Receive Message Queue
Data 7	9	Data = Cmd Code = Master Write-Read I^2C = 52h	
Data 8	10	Data = Bus ID = 01b	Private Bus of SMC
Data 9	11	Data = Slave Address* of FRU (in bits 7:1) = ACh (example)	
Data 10	12	Data = Number of bytes to read = 2 (example)	Byte count to read from FRU
Data 11	13	Data = Data to write = 13h (example)	Addr within FRU. Start addr for reading data.
Data 12	14	Check2 = checksum of 7 preceding bytes	End of encapsulated command sequence; also End of Master Write-Read I^2C command.

NOTE:
*Slave Address is obtained from by system management software from the Device Record Locator SDR for the SEEPROM.

The following is the response for the Send Message command.

Table 19.2-10: Ex D - RESPONSE Data for "Send Message" with encapsulated "Write-Read I^2C"

Field	byte #	Field Content	Comment
netFn / rqLUN	1	App Rs / LUN 00b = 000111b / 00b = 1CCh	BMC LUN 00b
Cmd Code	2	Send Message = 34h	
Completion Code	3	00h	00h = OK

After receiving the above response to "Send Message," System Management Software waits until the response data from the SMC (such as FRU content) is in the Receive Message Queue, then performs step 2. (When a message is received into the Receive Message Queue, the BMC sets the "ATN" flag in the System Interface.)

Step 2: Get (Response) Message

System Management Software uses a Get Message command to get the response from the SMC from the Receive Message Queue.

Table 19.2-11: Ex D - REQUEST Data for "Get Message"

Field	byte #	Field Content	Comment
netFn / LUN	1	App Rq / LUN 00b = 000110b / 00b = 18h	BMC LUN 00b. LUN is always 00b when communicating with BMC from system side
Cmd Code	2	Get Message = 33h	

The response from the Receive Message Queue includes the response to the Get Message command followed by the FRU's response to the Master Write-Read I^2C command, as shown in the next table.

Table 19.2-12: Ex D - RESPONSE Data for "Get Message"

Field	byte #	Field Content	Comment
netFn / LUN	1	App Rs / LUN 10b = 000111b / 00b = 1Ch	LUN 00b is always used for System Interface.
Cmd Code	2	Get Message = 33h	
Completion Code	3	00h	OK.
Channel #	4	00h	SMS Messaging Channel
Data 1	5	netFN / rqLUN = App Req. / SMS Message Buffer LUN = 000111b / 10b = 1Eh	Beginning of data in the Receive Message Queue. This is the Response Data returned via the IPMB from SMC in response to the Get Message command sent in step 1 above, minus the leading Slave Address (rqSA = BMC) of the IPMB Response.*
Data 2	6	Check1 = checksum of the preceding byte and BMC Slave Addr (20h)*	BMC Slave Address is included in checksum although it has been stripped off from command sequence.*
Data 3	7	RsSA = 22h	SMC addr. (responder slave addr.)
Data 4	8	RqSeq /rsLUN = 110101b / 00b = D4h	00b = SMC LUN. LUN where the response came from.
Data 5	9	Cmd Code = Master Write-Read I^2C = 52h	Response to Master Write-Read I^2C command.
Data 6	10	Data = Completion Code = 00h	00h = OK. Completion Code for Master Write-Read I^2C command.
Data 7	11	Data byte returned from SEEPROM offset 13h = ABh (example)	Byte count assumed. First requested data byte.
Data 8	12	Data bytes returned from SEEPROM offset 14h = CDh (example)	Second requested data byte.
Data 9	13	Check2 = Checksum of the 6 preceding bytes	End of Response Data for Master Write-Read I^2C, returned from SMC and stored in Receive Message Queue in BMC

*NOTE: Receive Message Queue holds the entire message received on the IPMB, minus the first byte. This byte gets stripped off when the BMC receives the request. This byte is the BMC's slave address, so the value is already known.

19.2.6 Example E: Reading FRU directly on IPMB Bus, via System Interface

Refer to block (E) of *Figure 19-5: Different Implementations of FRUs.*

This example reads 2 bytes of data beginning at address 13h in the FRU Inventory implemented as a non-intelligent device on the IPMB. To access a non-intelligent device on the IPMB, System Management Software sends a Master Write-Read I^2C command to the BMC. The command sequences are identical to Example (B) except for the Bus ID and Slave Address (rsSA) of the FRU. Differences are *italicized*.

Table 19.2-13: Ex E - REQUEST Data for "Master Write-Read I^2C"

Field	byte #	Field Content	Comment
netFn / rsLUN	1	App Rq / LUN 00b = 000110b / 00b = 18h	BMC LUN 00b
Cmd Code	2	Master Write-Read I^2C = 52h	
Data 1	3	Bus ID = 00b	BMC Public Bus (IPMB)
Data 2	4	*rsSA = Slave Address of FRU (in bits 7:1)* *= A8h (example)*	
Data 3	5	Number of bytes to read = 2 (example)	
Data 4	6	Data to write = 13h (example)	Offset within FRU (SEEPROM) to read from

Table 19.2-14: Ex E - RESPONSE Data for "Master Write-Read I^2C"

Field	byte #	Field Content	Comment
netFn/rqLUN	1	App Rs / LUN 00b = 000111b / 00b = 1Ch	BMC LUN 00b
Cmd Code	2	Master Write-Read I^2C = 52h	
Data 1	3	Completion Code = 00h	00h = OK
Data 2	4	Data byte read from FRU Addr 13h = ABh (example)	
Data 3	5	Data byte read from FRU Addr 14h = CDh (example)	

A. EXAMPLE: HARDWARE IMPLEMENTATION

The following is an example of an IPMI-based server management subsystem.

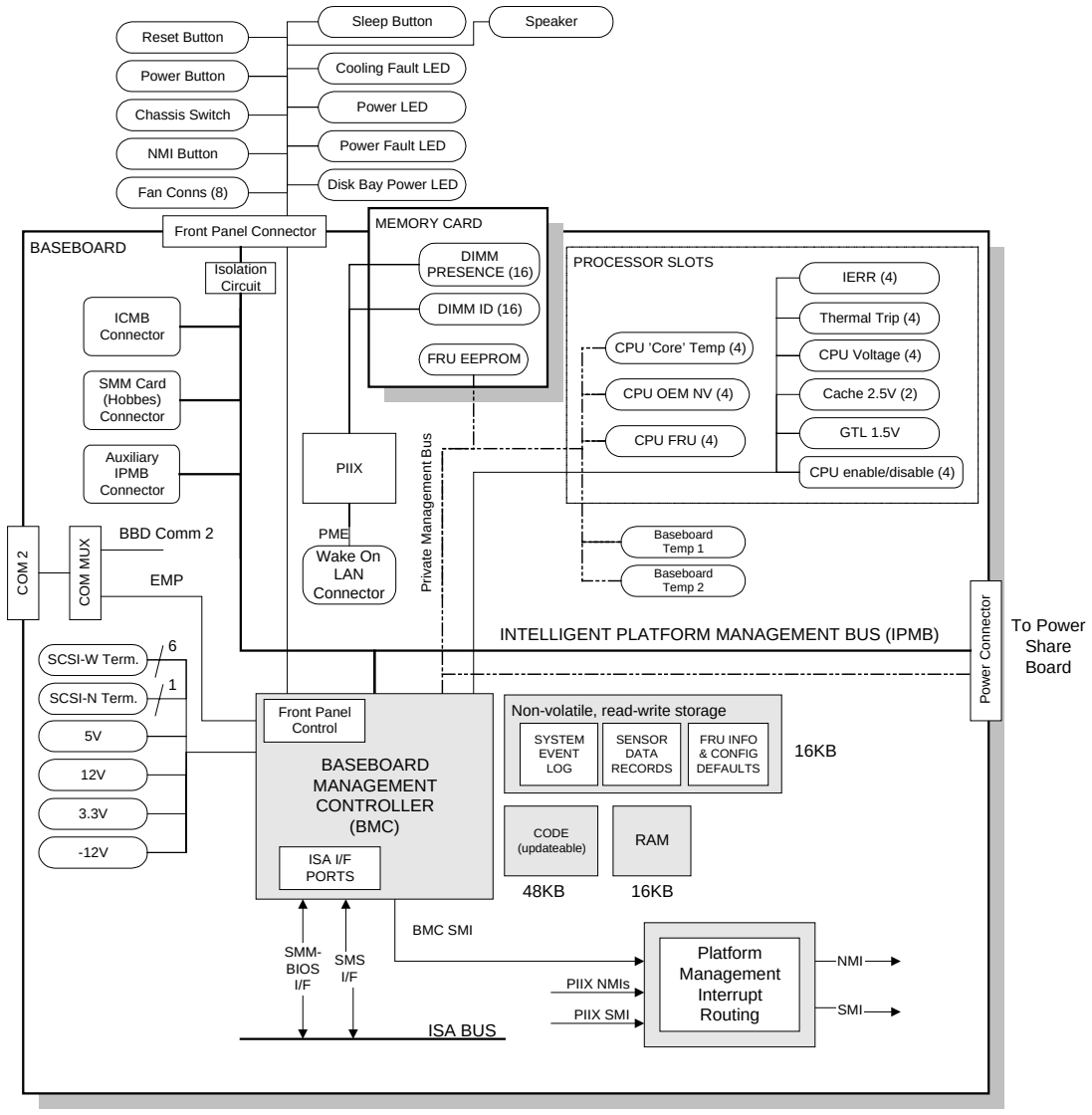


Figure 0-1: IPMI Hardware Example

B EXAMPLE: FRU LOAD FILE

The following is a sample FRU Load file:

```
// $Workfile:   nsbmc.fru  $
// $Author:   cjchapma  $
// $Revision:   1.2  $
// $Date:   23 Dec 1997 13:53:58  $
// $Modtime:   23 Dec 1997 11:17:38  $
// $Log:   R:/FW/NGHTSHDB.PRJ/NSBMC.FRV  $
//
//   Rev 1.2   23 Dec 1997 13:53:58   cjchapma
//_LF_VERSION    “0.16'
//- has &D0 in modem init string
//- has correct checksum for FRB area
//
//
//   Rev 1.1   19 Dec 1997 17:19:46   cjchapma
//    NASTOR15.FRU
//
//   Rev 1.0   17 Oct 1997 16:25:50   jbowker
//

_LF_NAME       “NightShade Baseboard”  // the name for this load file
_LF_VERSION    “0.16”                  // the version of this load file
_LF_FMT_VER    “1.3”                   // the version of the load file format
_SDR_VERSION   0100                    // Specifies the SDR format version #

_FRU (                        // FRU data for the Baseboard
_START_ADDR    0000           // Start address is 00h
_DATA_LEN      011D           // FRU Length is 285 bytes
_NVS_TYPE      “IMBDEVICE”    // AT24C64 for BMC
_DEV_BUS       FF             // ISA bus
_DEV_ADDRESS   20             // Device = BMC

_SEE_COMMON // NVS Common Header area contents 00-07h (8 bytes)
   01       // Common Header Format Version
   01       // Internal Use Area Offset in multiples of 8 bytes
   0E       // Chassis Information Area Starting Offset in multiples of 8 bytes
   12       // Board Area Starting Offset in multiples of 8 bytes
   1A       // Product Information Area Starting Offset in multiples of 8 bytes
   22       // Multi-Record Information Area Starting Offset in multiples of 8
bytes
   00       // PAD (1 bytes)
   A2       // Common Header Checksum (2's complement)

_SEE_INTERNAL        // NVS Internal Use area contents 08h-6Fh (104 bytes)
   01# 00# 00# 00# 00# 00# 00# 00# 00# 00# 00# 00# 00# 00# 00# 00# // 08h - 17h
   00# 00# 00# 00# 00# 00# 00# 00# 01# 20# 01# 20# 00# 00# 00# 00# // 18h - 27h
   00# 00# 00# 00# 00# 00# 00# 00# 0E# 00# 0E# 00# 00# 00# 00# 00# // 28h - 37h
   00# 00# 00# 00# 13# 88# 00# 80# 00# 80# FE# 64# 41# 54# 26# 46# // 38h - 47h
   30# 53# 30# 3D# 31# 53# 31# 34# 3D# 30# 26# 44# 30# 00# 00# 00# // 48h - 57h
   00# 00# 00# 00# 41# 54# 48# 00# 00# 00# 00# 00# 2B# 2B# 2B# 00# // 58h - 67h
   00# 00# 00# 00# 00# 00# 00# 00#                                 // 68h - 6Fh

_SEE_CHASSIS         // NVS Chassis Info area contents 70h-8Fh (32 bytes)
   01                // Chassis info area format version
   04                // Chassis info area length in multiples of 8 bytes
   11                // Chassis type (Main server chassis)
   CA                // 11 001010 Chassis part number type/length byte
   “000000-000”      // 10-digit part number as 8-bit ascii
   45                // 01 000101 Chassis serial number type/length byte
   99 12 34 56 78    // dummy 10 digit bcd serial number
   C1                // no more fields type/length byte
   // load utility 0 fills to end of area and inserts checksum
```

```
 _SEE_BOARD           // NVS Board Info area contents 90h-CFh  (64 bytes)
    01                // Board info area format version
    08                // Board info area length in multiples of 8 bytes
    00                // Unicode country base (00 = use 8-bit latin 1 ASCII)
    00 00 00          // dummy mfg date / time
    C5                // board mfr type/length byte
    "Intel"           // board manufacturer name
    C6                // product name type/length byte
    "N440BX"          // product name
    CA                // 11 001010 board serial number type/length byte
    "0123456789"      // dummy 10 digit ASCII board serial number
    CA                // 11 001010 board part number type/length byte
    "000000-000"      // 10-digit part number as 8-bit ASCII
    C1                // no more fields type/length byte

  _SEE_PRODUCT        // NVS Product Info area contents D0h-10Fh (64 bytes)
    01                // Product info area format version
    08                // Product info area length in multiples of 8 bytes
    00                // Unicode country base (00 = use 8-bit latin 1 ASCII)
    C5                // product mfr type/length byte
    "Intel"           // product manufacturer name
    CA                // product name type/length byte
    "NA440BX DP"      // product name
    C0                // 11 001100 product part number type/length byte
    C0                // null product version field type/length byte
    CA                // 11 001010 board serial number type/length byte
    "0123456789"      // dummy 10 digit ASCII product serial number
    C0                // null asset tag field type/length byte
    C1                // no more fields type/length byte

  _SEE_MULTIREC       // NVS MultiRecord Info area contents 110h-11Ch (13 bytes)
    00                // Record 1, Record Type ID
    80                // Record 1, area length in multiples of 8 bytes
    01                // Record 1,
    00                // Record 1, checksum of data
    7F                // Record 1, checksum of header
    00 00 00 00 00 00 00 00 // Record 1 data

 )  // End of FRU for BMC
```